Universidade Federal de Santa Catarina
Departamento de Informática e Estatística

Sistema de Automação Residencial baseado em Raspberry Pi e Open ZWave

Autor: Matheus Hoffmann Silva
Orientador: Prof. Dr. João Cândido Dovicchi

Florianópolis - SC
Novembro / 2014

Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
Graduação em Sistemas de Informação

Sistema de Automação Residencial baseado em Raspberry Pi e Open ZWave

Matheus Hoffmann Silva

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Florianópolis
2014

MATHEUS HOFFMANN SILVA

SISTEMA DE AUTOMAÇÃO RESIDENCIAL BASEADO EM RASPBERRY PI E OPEN ZWAVE

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Aprovada em Novembro de 2014.

BANCA EXAMINADORA

Prof. Dr. JOÃO CÂNDIDO DOVICCHI – Orientador
INE / CTC / UFSC

Prof. Dr. JOÃO BOSCO DA MOTA ALVES
INE / CTC / USFC

Prof. Dr. JEAN EVERSON MARTINA
INE / CTC / UFSC

Florianópolis-SC
2014

Lista de Figuras

2.1 Visão geral de uma rede de Automação Residencial. 5

3.1 Visão geral do eaZy. 14
3.2 Arquitetura v3 do eaZy 18
3.3 Raspberry Pi Modelo B. 24
3.4 Controlador ZWave: Aeotec Z-Stick S2. 25
3.5 Multisensor Homeseer HSM 100-S3. 25
3.6 Aeon Labs DSC06106-ZWUS Smart Energy Switch. 26
3.7 Modelo de Comunicação por Barramento. 28
3.8 Estrutura do componente Home Stack. 29
3.9 Estrutura do componente Device Controller. 30
3.10 Estrutura do componente eaZy Web. 32
3.11 Diagrama de fluxo da Interface de Usuário do eaZy. 33
3.12 Diagrama de Estados de um dispositivo no sistema. 34
3.13 Diagrama de Sequência da funcionalidade de listagem de dispositivos. 35

4.1 Dispositivos utilizados para o desenvolvimento do eaZy 37
4.2 Captura de Tela Desktop do *Dashboard* do eaZy. 38
4.3 Captura de Tela Dispositivo Móvel do *Dashboard* do eaZy. 39
4.5 Captura de Tela Dispositivo Móvel da Lista de Dispositivos eaZy. 39
4.4 Captura de Tela Desktop da Lista de Dispositivos do eaZy. 40
4.6 Captura de Tela Desktop do Gerenciador de Cenários do eaZy. 41
4.7 Captura de Tela Desktop do Formulário de Criação de um novo Cenário do eaZy. 42
4.8 Captura de Tela Dispositivo Móvel do Gerenciador de Cenário do eaZy. 42
4.9 Captura de Tela Desktop de Informações sobre Consumo do eaZy. 43
4.10 Gráfico de *commits* por semana do desenvolvimento do eaZy. 43
4.11 Linguagens de programação utilizadas no eaZy. 44

Lista de Tabelas

2.1 Comparação das tecnologias de comunicação em dispositivos de automação residencial. . . 12

3.1 Cronograma de Execução 17
3.2 Artefatos de Software 23
3.3 Especificação de Hardware do Raspberry Pi Modelo B. 23
3.4 Custos de Aquisição dos Artefatos de Hardware 26

Sumário

Lista de Figuras iii

Lista de Tabelas iv

1 Introdução 1

1.1 Contextualização . . . 1
1.2 Motivação . . . 1
1.3 Contexto da Aplicação . . . 2
1.4 Resultados Esperados . . . 2
1.5 Objetivos . . . 3
1.5.1 Objetivo Geral . . . 3
1.5.2 Objetivos Específicos . . . 3
1.6 Estrutura do Trabalho . . . 3

2 Referencial Teórico 4

2.1 Automação Residencial . . . 4
2.1.1 Definição . . . 4
2.1.2 Breve histórico . . . 4
2.1.3 Visão Geral de um Sistema de Automação Residencial . . . 5
2.1.4 Aplicações . . . 6
2.1.5 Desafios . . . 6
2.2 Trabalhos Relacionados . . . 7
2.2.1 Sistemas Proprietários . . . 7
2.2.2 Projetos de Código Aberto . . . 9
2.3 Tecnologias de Comunicação entre Dispositivos de Automação Residencial . . . 10
2.3.1 Discussão sobre a tecnologia a ser utilizada . . . 12

3 Metodologia 13

3.1 Projeto . . . 13
3.1.1 Visão Geral . . . 13
3.1.2 Riscos . . . 14
3.1.3 Requisitos . . . 15

3.1.4 Cronograma 17

3.1.5 Arquitetura 18

3.2 Subsídios Tecnológicos 19

3.2.1 Software 20

3.2.2 Hardware 23

3.3 Processo de Desenvolvimento e Preparação 26

3.3.1 Dependências 26

3.3.2 Ferramental de Apoio 27

3.3.3 Desenvolvimento para Hardware Embarcado 27

3.4 Implementação 27

3.4.1 Comunicação entre Componentes 27

3.4.2 Componente Home Stack 29

3.4.3 Componente Device Controller 30

3.4.4 Componente de Interface com Usuário – eaZy Web 31

3.5 Interação entre componentes 33

4 Resultados Obtidos **36**

4.1 Código Fonte 36

4.2 Dispositivos de Automação Residencial Utilizados 36

4.3 Demonstração do Sistema 36

4.4 Dados sobre o Desenvolvimento do Sistema 43

5 Conclusões **45**

5.1 Discussão 45

5.1.1 Dificuldades e soluções 45

5.1.2 Avaliação da Solução 46

5.2 Trabalhos Futuros 46

5.3 Conclusão 47

Referências Bibliográficas **48**

Anexos **51**

A Scripts utilizados **52**

A.1 Instalação de dependências 52

A.2 Variáveis de ambiente 54

A.3 *Deploy* do sistema 54

A.4 Inicialização do sistema 55

B Código Fonte **57**

B.1 Biblioteca de Comunicação por Barramento 57

B.2 Componente Home Stack 63

B.3 Componente Device Controller 77
B.4 Componente eaZy Web 83

C Artigo **88**

Capítulo 1

Introdução

1.1 Contextualização

Com evolução tecnológica intensa e avanços na área de Interação Humano-Computador (HCI), cada vez mais são criados novos meios de comunicação e possibilidades de interconexão e troca de informações entre máquinas e pessoas. A fim de trazer benefícios para seu dia-a-dia, é comum o homem incluir tecnologia na sua vida. Seja por questão de comodidade ou necessidade, a 'Internet das Coisas'[10] têm invadido o cotidiano e inovado a maneira como as pessoas interagem entre si e o meio em que elas vivem. Nesse sentido, a Automação Residencial é a tecnologia que visa melhorar a qualidade de vida de pessoas na suas casas, seja auxiliando-as na realização de tarefas rotineiras ou tornando o ambiente mais seguro e confortável o possível, de maneira automatizada.

Uma solução de Automação Residencial faz uso de tecnologias de automação (*eg.* sensores de presença ou de luz) para reconhecer o estado do ambiente no qual está inserido e, com o auxílio de configurações especificadas pelo seu usuário, faz uso desta informação para agir neste ambiente. Um possível cenário de utilidade de um sistema de Automação Residencial seria notificar os pais quando seus filhos chegaram em casa da escola, ou ainda, com um só toque no *Smart Phone*, desligar as luzes e trancar as portas e janelas da residência quando o usuário se prepara para dormir.

Ao estudar as soluções de Automação Residencial já existentes e as tecnologias utilizadas para essa automação, o presente trabalho pretende explorar e propor uma solução que atenue dois dos principais fatores que ainda inviabilizam ou dificultam a aquisição de sistemas de Automação Residencial para muitos usuários [2]: o alto custo de aquisição e complexidade de instalação e configuração do sistema.

1.2 Motivação

Atualmente existem diversas soluções de Automação Residencial, tanto proprietárias, como o Fibaro [4] e Control4 [7], quanto soluções de código aberto, como Linux MCE [8] e MajorDoMo [9].

Na busca por uma solução que se encaixe nas suas necessidades, o utilizador pode optar por uma das soluções de código aberto, que não necessitam de um gasto para aquisição uma licença de

software. Apesar de vários destes sistemas apresentarem características inovadores e constituírem soluções completas, exigem configuração complexa e pressupõem que o instalador, por serem soluções DIY (*do-it-yourself* ou 'faça você mesmo'), tenha conhecimento técnico para configurar e integrar corretamente a rede de dispositivos do sistema. Além disso, há sistemas que apresentam interface pouco amigável, que dificultam seu uso. Estes fatores contribuem para a possibilidade de uma experiência frustrante por parte do utilizador.

No caso de sistemas proprietários, existe um custo bastante elevado para aquisição, instalação e manutenção do sistema, tornando esse tipo de solução inviável para usuários que não desejam investir grande quantidade de recurso financeiro na automação de sua residência.

O presente trabalho foca no desenvolvimento de uma solução de Automação Residencial de código aberto de fácil configuração, sem necessidade de gastos com aquisição de licenças de software e de arquitetura flexível o bastante para utilizar um variado leque de tecnologias de comunicação de dispositivos: o eaZy.

1.3 Contexto da Aplicação

O eaZy, sistema proposto neste trabalho, é uma solução de Automação Residencial modelo DIY (*ie.* 'faça você mesmo') de código aberto, cujo principal objetivo é possibilitar a automação de uma residência sem a necessidade de complexas configurações e instalações intrusivas. O sistema não é feito para competir com as grandes soluções disponíveis no mercado, mas sim prover automação simples e pontual.

O público alvo do eaZy são pessoas que desejam implementar automatização simples e efetiva em suas residências sem necessidade de investimentos financeiros elevados ou mudanças estruturais (*eg.* instalação de cabeamentos).

1.4 Resultados Esperados

É esperado que o resultado deste trabalho seja um sistema de Automação Residencial com as funcionalidades principais para que seja possível realizar a automação simples de uma residência, utilizando pelo menos um protocolo de comunicação com dispositivos de automação, e com pelo menos uma interface de usuário.

Como resultado esperado deste estudo, consideram-se:

- Integração com uma tecnologia de comunicação de dispositivos de automação.
- Controle de dispositivos (*eg.* tomadas, lâmpadas).
- Ter informações atualizadas de sensores.
- Possibilidade de configuração de cenários simples e funcionais, definindo ligações entre pelo menos dois dispositivos.
- Pelo menos uma interface de usuário.

Os seguintes ítens não são esperados como resultado final deste estudo:

- Produto de automação residencial completo e competitivo com soluções de mercado.
- Integração com mais de uma tecnologia de comunicação de dispositivos de automação.

1.5 Objetivos

1.5.1 Objetivo Geral

Desenvolver uma solução de automação residencial de código aberto que seja diferenciada pela sua simplicidade de instalação e configuração, além de ter custo tão baixo quanto possível, para que seja possível que usuários com pouco conhecimento técnico das tecnologias utilizadas possam instalar e manter o sistema.

1.5.2 Objetivos Específicos

Durante o desenvolvimento do sistema proposto, os seguintes passos serão seguidos:

- Estudos sobre Automação Residencial: definição, visão geral e histórico.
- Estudos sobre as Tecnologias e Trabalhos Relacionados e definição das Tecnologias de Automação Residencial a serem utilizadas.
- Projeto e Arquitetura do eaZy.
- Implementação do eaZy.
- Avaliação dos resultados.

1.6 Estrutura do Trabalho

Este trabalho foca no desenvolvimento de uma solução de Automação Residencial que apresente um impacto mínimo de instalação, sem necessidade de mudanças estruturais na casa do usuário, e prezando ao máximo pela facilidade de configuração e baixo custo dos dispositivos que compõem a rede do sistema. A aplicação a ser desenvolvida não se propõe a superar outras soluções existentes em termos de automação e funcionalidades, mas sim focar em um nicho específico, que é a automação residencial extensível de baixo custo.

A estrutura do trabalho divide-se da seguinte maneira: o Capítuo 2 (Referencial Teórico) traz um estudo sobre o que é, breve histórico, aplicações e desafios de Automação Residencial além de estudar e comparar soluções e tecnologias de Automação Residencial já existentes no mercado. O Capítulo 3 (Metodologia) discorre sobre o projeto, definições de arquitetura e tecnologias, o hardware utilizado, e a implementação da solução. No Capítulo 4 (Resultados Obtidos) são demonstrados os resultados do esforço de implementação da solução, e no Capítulo 5 (Conclusão) são discutidas características que podem ser futuramente agregadas ao sistema e dadas as considerações finais.

Capítulo 2

Referencial Teórico

2.1 Automação Residencial

2.1.1 Definição

Automação Residencial, *Home Control Systems* (HCS) ou ainda Domótica (no latim, domo é *casa*) [3] é 'o sistema que remove tanta interação humana quanto possível e desejável de tarefas domésticas, substituindo-a por sistemas eletrônicamente automatizados', com o objetivo de tornar o dia-a-dia dos ocupantes da casa mais eficiente e confortável.

Alguns exemplos possíveis de automação:

- Receber uma notificação no celular quando os filhos chegam em casa.
- Desligar luzes, trancar portas e ativar sistemas de segurança com apenas um botão.
- Fechar as janelas da casa ao detectar presença de chuva.

2.1.2 Breve histórico

Há décadas existem estudos e soluções de Automação Residencial. A primeira solução documentada é o ECHO IV [1]: em 1966, o engenheiro norte-americano Jim Sutherland criou o espaçoso modelo experimental ECHO IV (*Electronic Computing Home Operator*), um sistema de automação caseira multitarefas. O sistema era capaz de fazer a contabilidade financeira da família e, além disso, servia como uma central de mensagens, onde os familiares poderiam deixar recados de uns para os outros. Apesar de nunca ter sido produzido comercialmente, o ECHO representou um grande avanço em tecnologias de automação da época, por ter sido o primeiro a ser utilizado no ambiente de casa e ser capaz de ajudar seus ocupantes.

Em 1969, a compania Honeywell lançou o H316 [18], conhecido como Kitchen Computer (computador de cozinha), que foi o primeiro computador caseiro comercializável. O H316 podia ser programado para armazenar receitas e dar dicas de culinária. Apesar da inovação, o Kitchen Computer foi um fracasso devido ao seu custo benefício, pois o público consumidor não achava justificável o absurdo preço de dez mil dólares que custavam o H316.

Nas décadas seguintes houveram grandes evoluções das tecnologias que seriam, mais tarde, essenciais para a Automação Residencial. Estas evoluções possibilitaram a miniaturização e potencialização de chips e circuitos, de modo que fosse possível adicionar soluções de automação inteligentes e de pequeno porte às tomadas, lâmpadas e sensores.

Entre as principais áreas tecnológicas que servem de base para as tecnologias de automação residencial, pode-se citar: robótica, que permite transcender a divisão entre mundo físico e virtual, de modo que comandos de computador possam se materializar em ações no ambiente; inteligência artificial, que é essencial para construção sistemas adaptativos que possam evoluir e responder às mudanças do ambiente, de modo que a atuação do sistema é consistente com a necessidade pontual do usuário; e convergência, também conhecida como 'Internet das coisas'(IoT) [10], que permite uma comunicação transparente entre computadores, dispositivos e sensores de modo a possibilitar novos meios de interação.

2.1.3 Visão Geral de um Sistema de Automação Residencial

Para que um sistema de automação caseira possa cumprir seu objetivo, este precisa ter informações sobre o ambiente em que se encontra. Com base nessas informações e de cenários pré-programados, é possível que o sistema execute ações no ambiente residencial para chegar à um estado desejado. Um exemplo deste fluxo:

Num ambiente hipotético, existe um sensor de presença. O sistema de automação do qual o sensor faz parte foi programado para ligar lâmpadas caso alguma presença seja detectada. Então, quando o sensor detecta presença, envia um sinal para o sistema. Quando o sistema é alertado sobre a presença no ambiente, checa se há ações configuradas para este evento, e então manda um sinal para as lâmpadas se acenderem.

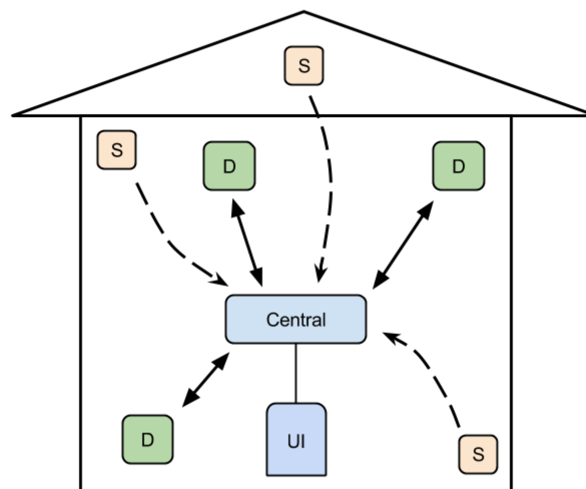


Figura 2.1: Visão geral de uma rede de Automação Residencial.

Uma visão geral de um um sistema de Automação Residencial pode ser visualizado na Figura 2.1. Para coletar dados sobre um ambiente o sistema conta com sensores (*eg.* presença, temperatura e luminosidade), que são representados na Figura 2.1 como 'S'. Estes sensores constantemente se comunicam

com a central do sistema para mantê-lo atualizado.

No momento em que o sistema necessita realizar uma ação, seja por interação direta do usuário ou por um cenário pré-programado, faz uso dos dispositivos de automação. Estes dispositivos, representados como 'D'na Figura 2.1, possibilitam o controle e coleta de dados de eletrodomésticos, luzes, sistema de som, entre outros.

A manutenção e configuração do sistema é feita por meio de uma Interface de Usuário, representada como 'UI'na Figura 2.1, que pode ser, por exemplo, por meio de uma aplicação para telefone móvel ou navegador de internet.

2.1.4 Aplicações

Automação Residencial pode trazer muitos benefícios para o ambiente onde é aplicada, seguem alguns exemplos de cenários de utilização:

- conforto: um sistema de automação pode ter foco no conforto e entretenimento dos seus usuários. Neste caso, o sistema pode ser integrado aos sistemas multimídia da residência, possibilitando ao usuário controle sobre suas mídias, além de controle de climatização e iluminação da residência;

- segurança: portas, janelas e portões podem ser automatizados e junto com sensores de presença e câmeras podem constituir num sistema de segurança completo. É possível que o usuário receba notificações caso, por exemplo, saia de sua residência e esqueça portas abertas e controlá-las remotamente;

- saúde: um sistema de automação residencial pode ajudar pessoas desabilitadas ou idosos facilitando sua interação com a casa, de modo que ajustes de climatização, segurança, alarmes e sistemas de entretenimento sejam gerenciados por meio de um interface compatível com suas desabilidades (*eg.* comandos de voz, controles remotos).

2.1.5 Desafios

Um grande desafio na área de Automação Residencial e alvo de vários estudos [19] [20] [21], é a aplicação de modelos baseados em inteligência artificial para tornar a residência realmente inteligente, uma *Smart Home* (casa inteligente). Muitos trabalhos relatam desenvolvimento e estudo de soluções de Automação Residencial utilizando a denominação *Smart Home*. A denominação casa inteligente pode ser considerada precipitada, pois o termo '*implicita que a casa se adapta aos seus habitantes*'[2], de modo que '*a inteligência do sistema surge da habilidade da casa de predizer o comportamento e necessidades de seus habitantes após um período de tempo de observação*'[19]. Ou seja, uma *Smart Home* é automatizada sem intervenção de seus habitantes, e executa ações no ambiente de acordo com o perfil observado dos seus usuários. O termo que melhor representa as soluções disponíveis até o momento, e também a solução proposta neste trabalho, é Automação Residencial.

Outro desafio é a busca por um modo de interação livre entre o usuário e a casa. Os modos mais comuns de interação são computadores, interfaces sensíveis ao toque, aplicativos para celular ou

ainda pela televisão. Estes meios apresentam limitações de uso e pressupõe que o utilizador deve estar em contato direto com o dispositivo em questão para interagir com o sistema. Existem casos nos quais o utilizador necessita de uma interação mais livre e sem contato direto com dispositivos, como por exemplo pessoas com deficiências ou de idade avançada. Exemplos de soluções que apresentam novas formas de interação: Controle por meio de gestos é proposto em [22], onde o usuário empunha um controle WiiMote (Nintendo) para realizar gestos, mas o sistema apresenta dificuldades para definir qual o dispositivo que é alvo de atenção do usuário. Controle por gestos via telefone móvel (*smart phone*) também é proposto em [25], que demonstra uma boa taxa de reconhecimento dos gestos, mas estes nem sempre são intuitivos os bastante para representar a ação desejada. Controle por comandos de voz é proposto em [23], onde uma série microfones são usados para capturar a voz do usuário e traduzir em comandos para a casa, mas o vocabulário reconhecido pelo sistema ainda é muito limitado e as ocasionais interferências sonoras dificultam a interpretação apurada do comando falado.

A maneira como o usuário define os cenários de automação de sua casa também é um desafio. A configuração de cenários não triviais, como por exemplo 'abrir as janelas da sala se for dia e não estiver chovendo', quando possível, exige certo conhecimento técnico por parte do usuário. Há estudos de soluções que focam em simplificar a definição de cenários de Automação Residencial como por exemplo [24], que é uma interface de programação para usuários finais. O projeto permite ao utilizador definir cenários utilizando uma interface baseada em ímãs de geladeira virtuais (*drag and drop*) para organizar o cenário desejado, que então é processado pelo sistema e transformado em linguagem de código interpretável por um sistema de Automação Residencial.

2.2 Trabalhos Relacionados

O estudo de sistemas relacionados, ou seja, soluções de automação residencial já existentes, é importante para definir as características de soluções proprietárias e de código aberto para que, assim, haja subsídios para justificar a criação de um novo sistema que possa suprir um nicho não devidamente explorado.

2.2.1 Sistemas Proprietários

Fibaro Home Intelligence

O Fibaro Home Center 2 [4] é uma solução proprietária desenvolvida pela empresa polonesa Fibar Group, que utiliza ZWave como tecnologia de comunicação. Como o ZWave é uma tecnologia baseada em frequência de rádio, o sistema é flexível quanto à distribuição de seus dispositivos na casa que não necessitam de fios para comunicação com a central, o que apresenta a vantagem de uma instalação não instrusiva.

O Home Center 2 faz uso de uma central de controle, que conta com um *set-top* próprio. Esta central fica responsável por todo o monitoramento e controle dos dispositivos da casa. Além da automação de dispositivos, a solução oferece integração multimídia, possibilitando utilizar o hardware do Home Center 2 como gerenciador de bibliotecas de músicas e vídeos do utilizador.

O principal meio de configuração da solução é por meio de um navegador de internet. A central provê uma interface *web* por meio da rede local WiFi (IEEE 802.11g) que pode ser acessada por qualquer dispositivo com acesso à rede da casa. Além da interface *web*, há interface para televisão, aplicativo para telefone móvel e notificações via correio eletrônico e mensagens de celular.

A característica mais interessante do Home Center 2 é a interface gráfica para configuração cenários. A interface apresenta os dispositivos da rede e dá um conjunto de variáveis e estruturas de controle para que seja possível configurar a cena desejada visualmente.

A principal limitação deste sistema é o fato de aceitar apenas dispositivos ZWave, o que significa que se usuários que já possuem dispositivos que utilizam outras tecnologias não poderão utilizá-los ao adquirir o Home Center 2. Além disso, é preciso adquirir o hardware da Fibaro, sem possibilidade de utilizar a solução em outra plataforma.

Home Control Assitant

O Home Control Assistant [5], desenvolvido pela empresa americana Advanced Quonset Technology, é uma solução de Automação Residencial que aposta na compatibilidade de vários sistemas de comunicação para ter flexibilidade como seu diferencial, além de uma grande quantidade de funcionalidades. Entre as tecnologias suportadas pelo HCA, com algumas limitações, estão Insteon, UPB, ZWave e X10.

O o sistema possibilita a automação completa e controle de dispositivos como luzes e tomadas inteligentes, com integração de sensores. O foco do sistema é automação, o HCA não tem opções multimídia. Existe uma interface de configuração de cenários, que recebe o nome de 'programas', que fornece opções de controle de acordo com horários definidos pelo usuário ou eventos capturados pelos sensores da rede de dispositivos.

Uma limitação do HCA é o fato de ser disponibilizado apenas para a plataforma Microsoft Windows, e o fato da sulução necessitar que o usuário tenha um computador com este sistema operacional em sua residência. Além disso, o computador onde é instalada a solução, que frequentemente é o computador pessoal de uso diário, deve sempre estar ligado para que o HCA permaneça operante.

A interação com o sistema acontece, principalmente, por meio do software instalado no Desktop Windows do usuário, mas também há aplicações simples para as plataformas de telefones móveis Android e iOS.

O usuário que deseja adquirir a solução deve escolher entre os três tipos de conta disponíveis, que possibilitam acesso extremamente limitado até completo do sistema. Além disso, precisa ter um computador com sistema operacional Microsoft Windows, e uma rede dispositivos para que seja possível a automação de sua residência.

Control4

O Control4 [7] é na verdade um conjunto de soluções de automação residencial, que utilizam sua plataforma de *Smart Home*. As soluções tem foco variado, de sistemas de segurança com fechaduras automatizadas, câmeras e sensores até sistemas multimídia com possibilidade de reproduzir música em qualquer ambiente da casa.

Os dispositivos e controladores utilizados por estas soluções são proprietários da Control4, o que traz a desvantagem de que outros dispositivos são incompatíveis com o sistema.

A principal vantagem do Control4 é a integração plena entre suas soluções, sendo que na mesma interface é possível que o usuário tenha controle sobre o ambiente de sua residência, em termos de temperatura, luminosidade e segurança, além de, por exemplo, possibilidade de controlar os sistemas multimídia espalhados pela casa.

O sistema também possibilita a definição de cenários que, de acordo com pré-definições especificadas pelo usuário, automatizam certos eventos. Um exemplo de cenário possível com o Control4 é um alerta via mensagem de texto para os pais, indicando que seus filhos chegaram em casa.

O usuário do Control4 interage com o sistema de vários modos, como por exemplo controles remotos similares aos de aparelhos de televisão, telas sensíveis ao toque imbutidas na parede ou por meio de um navegador de internet.

2.2.2 Projetos de Código Aberto

Linux MCE

O Linux MCE [8] é um projeto de Automação Residencial completo e de código aberto, baseada no sistema operacional Ubuntu 8.10, uma distribuição Linux. Suporta várias tecnologias, entre estas ZWave, Insteon e X10.

Como pontos fortes do sistema, além de ser software livre, pode-se citar a flexibilidade em termos de tecnologia, que possibilita que dispositivos de diferentes protocolos de comunicação possam ser utilizados pela solução, e as várias características integradas ao sistema, que apresenta possibilidade de configuração de preferências de automação por usuário, biblioteca multimídia e sistema de chamada VoIP.

Como desvantagem, pode-se citar a interface desktop pouco amigável e a dificuldade de instalação e configuração do sistema, que assume que o instalador tem certo nível de conhecimento das tecnologias a serem utilizadas, como por exemplo do sistema operacional Linux.

MajorDoMo

O MajorDoMo (Major Domestic Module) [9] é uma plataforma de Automação Residencial de código aberto cujo objetivo é a facilidade de instalação e de configuração. O sistema suporta tecnologia ZWave para comunicação com dispositivos de automação, e é distribuído para as plataformas Microsoft Windows e Linux.

O sistema faz uso de uma linguagem de configuração visual que auxilia o usuário a definir um cenário sem necessidade de codificá-lo.

Uma desvantagem da solução é o fato de que a maior parte da documentação está apenas disponível em Russo, o que dificulta muito a utilização e configuração do sistema para não falantes.

2.3 Tecnologias de Comunicação entre Dispositivos de Automação Residencial

Nesta seção são estudadas algumas tecnologias utilizadas como protocolos de comunicação em sistemas de automação. O foco deste estudo são tecnologias compatíveis com a o sistema proposto neste trabalho, no qual uma das principais premissas é a facilidade de instalação de dispositivos. Por conta disso, não serão consideradas tecnologias que necessitem de instalação intrusiva ou de mudanças estruturais na casa do usuário, como por exemplo tecnologias que dependentes de ligações físicas extras (*eg.* fiação e cabeamento) para realizar a comunicação.

Seguindo esta premissa, serão expostas tecnologias que utilizam **(a)** A rede elétrica de fios já presente no domícilio, e **(b)** tecnologias de comunicação sem fio.

X10

O X10 é um padrão industrial de comunicação entre dispositivos eletrônicos criado em 1975, pela empresa escocesa Pico Electronics [11]. O padrão foi a primeira tecnologia de automação residencial acessível ao público, pelo fato de ter um custo relativamente baixo. O X10 utiliza a rede elétrica, infraestrutura já presente na casa, para sinalizar e controlar dispositivos. O protocolo se baseia em pulsos de rádio frequência para transmitir dados em *broadcast* pela rede elétrica.

As principais desvantagens do X10 são a sua velocidade de transmissão de dados, que é próxima de 20 bit/s, e a falta de suporte para criptografia. O X10 possibilita comunicação simultânea com no máximo 256 dispositivos.

O X10 é um protocolo já considerado ultrapassado mas muitos sistemas de automação residencial ainda mantém compatibilidade, levando em consideração os dispositivos legados que utilizam este padrão (*eg.* Home Control Assistant).

Universal Powerline Bus

O UPB – Universal Powerline Bus [6] é um protocolo de comunicação via rede elétrica baseado no X10, desenvolvido pela Powerline Control Systems em 1999. O padrão faz uso de pulsos elétricos temporizados, chamados de ‘Pulsos UPB’, que se sobrepõe em comparação às formas de onda AC da corrente elétrica, para comunicar dados digitalizados.

O UPB apresenta uma taxa de transmissão mais elevada que o X10, contando com 240 bit/s, além de aprimoramentos na confiabilidade do sistema. A topologia do UPB é flexível, funcionando de modo peer-to-peer com ligações simples entre dispositivos, ou utilizando uma central para controle de todos os dispositivos.

Insteon

Tecnologia criada com o objetivo de superar as limitações do X10 e adicionar adicionar flexibilidade. O Insteon [12], concebido em 2005 pela americana Smart Labs Inc, tem banda dupla e suporta comunicação

tanto via rede elétrica quanto via frequência de rádio. O protocolo apresenta compatibilidade com o padrão X10.

O Insteon tem topologia peer-to-peer, e a velocidade de transmissão de dados ultrapassa bastante seus predecessores, chegando a aproximadamente 12,8 kbit/s via rede elétrica e 37,5 kbit/s via ar.

ZigBee

O ZigBee [13] é um protocolo de comunicação que utiliza frequências de rádio para criar uma rede local, baseado no padrão IEEE 802.15.4-2006 [14], para disponibilizar um ambiente de comunicação sem fio para a rede de dispositivos de um sistema de automação residencial. Concebido em 1998, normatizado em 2003 e revisado em 2006, o ZigBee é mantido e publicado pela Zigbee Alliance.

A rede de dispositivos especificados pelo protocolo ZigBee necessita de um dispositivo controlador, responsável pela criação e manutenção dos parâmetros da rede. Este controlador pode ser o nodo central da rede ou não, dependendo da topologia de rede implementada.

O esquema de segurança da rede ZigBee origina do padrão IEE 802.15.4 e se baseia na troca de chaves simétricas. A velocidade de transmissão de dados é de até 250 kbit/s, e o alcance médio da rede é até 50 metros, dependendo dos dispositivos e do ambiente.

ZWave

Desenvolvido pela Zenzys Inc e lançado em 2004, o ZWave [15] é um protocolo inspirado no padrão ZigBee, que habilita a comunicação e controle de dispositivos por meio de frequência de rádio. Em 2008, a empresa norte-americana Sigma Designs adquiriu os direitos sobre o ZWave por meio da incorporação da Zenzys Inc.

A rede de dispositivos ZWave é de topologia de malha (ou *mesh*) [16], o que possibilita que um suposto dispositivo A pode se comunicar com outro dispositivo B, sem que B este esteja no raio de alcance de recepção de sinal de A, utilizando os dispositivos intermediários entre A e B. Os dispositivos ZWave podem receber, transmitir e repetir mensagens pela rede.

O ZWave necessita de um dispositivo central para gerenciar sua rede, chamado de controlador ZWave, que é a interface entre um sistema de automação residencial e sua rede de dispositivos ZWave. A rede pode conter até 232 nodos e a taxa de transmissão de dados é de até 100 kbit/s. Uma rede ZWave recebe um identificador único, chamado de 'Home ID', e cada dispositivo tem um identificador único e imutável chamado de 'Node ID'. Estes identificadores, aliados à métodos de criptografia e autenticação de dispositivos nas trocas de mensagens, atribuem à segurança da rede.

A Sigma Designs é proprietária da protocolo ZWave, que não disponibiliza sua especificação ao domínio público. A Sigma comercializa um 'Kit de Desenvolvedor', que possibilita implementar sistemas de automação residencial utilizando dispositivos ZWave, mas o uso do Kit exige também a aceitação de um acordo de não divulgação de detalhes sobre o protocolo.

Existem alternativas, desenvolvidas a partir de análize de tráfego de dados de dispositivos ZWave, para se construir sistemas de automação residencial sem a necessidade de adquirir o Kit de Desenvolvedor vendido pela Sigma. É o caso do **Open ZWave** [17], um projeto de código aberto que provê uma Interface

de Programação de Aplicação (API) que, ao ser incorporada à um sistema, permite que este seja capaz de se comunicar e gerenciar uma rede de dispositivos ZWave.

2.3.1 Discussão sobre a tecnologia a ser utilizada

Um resumo do estudo de tecnologias de comunicação de Dispositivos de Automação Residencial pode ser visualizado na Tabela 2.1. Enquanto cada tecnologia apresenta suas vantagens e desvantagens, ressaltam-se duas que aproximam-se mais do perfil do presente trabalho:

- o Insteon, que é um protocolo versátil e com dispositivos de fácil instalação, que utiliza tanto a infraestrutura de rede elétrica já presente na casa e rádio para comunicação, além de ser compatível com dispositivos X10;
- e o ZWave, que apresenta um grande número de dispositivos já presentes no mercado, facilidade de instalação, configuração e manutenção da rede de dispositvos, e segurança na comunicação que é exclusivamente via rádio.

Tecnologia	**Meio de Comunicação**	**Taxa de Transmissão de Dados**
X10	Rede Elétrica	20 bit/s
UPB	Rede Elétrica	240 bit/s
Insteon	Rede Elétrica, Rádio	Fio: 12,8 kbit/s, Ar: 37,5 kbit/s
ZigBee	Rádio	250 kbit/s
ZWave	Rádio	100 kbit/s

Tabela 2.1: Comparação das tecnologias de comunicação em dispositivos de automação residencial.

Capítulo 3

Metodologia

No capítudo Metodologia são descritos os processos de estudo, desenvolvimento e concepção do eaZy. De acordo com cada cada Objetivo Específico listado na Sessão 1.5.2, os ítens abordados:

- O ojetivo *Estudos sobre Automação Residencial: definição, visão geral e histórico* é abordado na Sessão 2.1 Automação Residencial.
- O objetivo *Estudos sobre as Tecnologias e Trabalhos Relacionados e definição das Tecnologias de Automação Residencial a serem utilizadas* é abordado na Sessão 2.2 Trabalhos Relacionados e na Sessão 2.3 Tecnologias de Comunicação entre Dispositivos de Automação Residencial. Além disso, foram realizados estudos para definição de Software e Hardware a serem utilizados pelo sistema na Sessão 3.2 Subsídios Tecnológicos.
- O objetivo *Projeto e Arquitetura do eaZy* é abordado na Sessão 3.1 Projeto.
- O objetivo *Implementação do eaZy* é descrito na Sessão 3.4 Implementação.
- O objetivo *Avaliação dos resultados* é abordado no Capítulo 4 Resultados Obtidos.

3.1 Projeto

Nesta sessão são estudados os aspectos de projeto do sistema proposto no presente trabalho: Visão Geral, Riscos, Requisitos, Cronograma e Arquitetura.

3.1.1 Visão Geral

O eaZy é um Sistema de Automação Residencial que tem por objetivo a automação simples de uma residência, para que seu utilizador tenha informações sobre seu ambiente domiciliar e tenha controle sobre ele mesmo remotamente. Os principais requisitos para a criação do sistema são: facilidade de configuração, utilização de hardware de baixo custo e arquitetura flexível.

O sistema é capaz de se comunicar com dispositivos que utilizam o protocolo de *ZWave*, de modo que é possível coletar informações e executar ações sobre eles. Por exemplo: é possível como controlar

remotamente tomadas, lâmpadas e coletar informações de dispositivos e sensores, como consumo de energia elétrica, temperatura e luminosidade.

A Figura 3.1 demonstra uma visão geral de como o eaZy é situado no ambiente de automação.

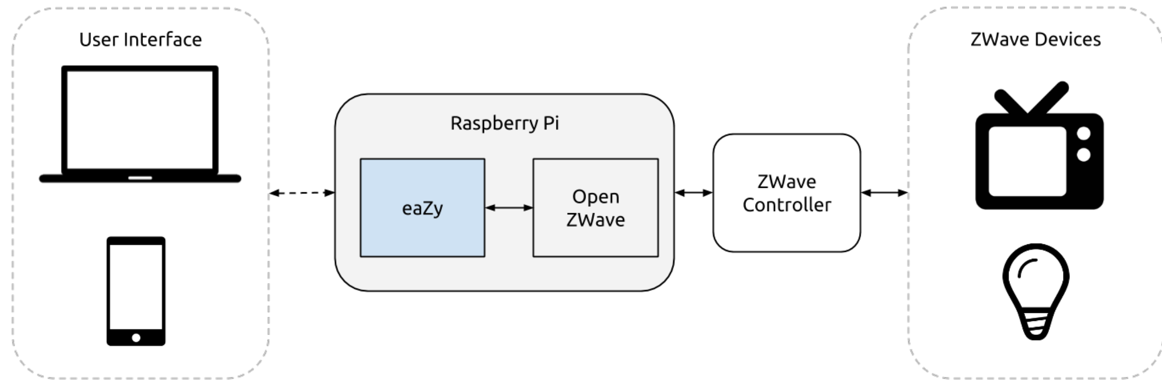


Figura 3.1: Visão geral do eaZy.

As principais funcionalidades do eaZy são o controle de dispositivos e a possibilidade de criação de cenários: o **controle de dispositivos** acontece por meio de uma lista de dispositivos no sistema. Esta lista reflete cada dispositivo presente na rede física de dispositivos *ZWave*, e por meio dela é possível visualizar dados de cada dispositivo e executar ações sobre eles (*eg.* ligar e desligar); e a funcionalidade de **definição de cenários**: o utilizador pode definir que uma ação ocorra automaticamente à partir de um determinado estado (*eg.* quando um sensor de presença detecta movimento, o sistema automaticamente liga as lâmpadas no cômodo).

A interação com o sistema acontece por meio de uma **interface web** simples e amigável, acessível via qualquer computador ou dispositivo móvel que disponibilize de um navegador de internet.

O eaZy é compatível com Sistema Operacional Linux e não necessita de hardware com alto nível de performance, de modo que pode ser instalado em computadores de baixo custo como o Raspberry Pi. Para que o sistema funcione normalmente, é necessário que o computador no qual foi instalado esteja conectado à uma rede de computadores. Então, por meio de sua interface *web*, é acesivel por outros computadores ligados à mesma rede.

A licença de software do eaZy é MIT, ou seja, seu código é aberto.

3.1.2 Riscos

Na concepção de um projeto de software, existem diversos riscos que podem ameaçar o seu sucesso. Assim, há necessidade de identificar os riscos do projeto eaZy e criar meios para que o risco não venha a ser uma ameaça real para o projeto. São listados a seguir os Riscos identificados para o projeto, assim como meios para mitigá-los:

Risco 1 Há a possibilidade de que o projeto sofra atrasos no cronograma, por conta de eventos inesperados ou erros de planejamento. Um atraso pode resultar na redução de escopo no sistema, finalizar o

projeto com sistema inacabado ou mesmo inoperante.

Mitigação: planejar o cronograma com cautela e ter uma boa definição do escopo do projeto.

Risco 2 Dentre os vários componentes que integram o eaZy há muitos artefatos de software utilizados para diferentes fins, então existe a possibilidade de incompatibilidade entre os artefatos de software, ou com o sistema operacional escolhido.

Mitigação: Efetuar levantamento sobre os artefatos de software e suas compatibilidades tanto entre si quanto com o Sistema Operacional.

Risco 3 Para auxiliar no desenvolvimento do sistema, é necessário que estejam disponíveis dispositivos reais utilizados para Automação Residencial, assim o sistema pode ser testado com casos de uso reais. O desenvolvimento do sistema com ausência de dispositivos pode resultar em falhas ou mesmo fracasso do projeto.

Mitigação: Adquirir dispositivos de Automação Residencial com andecedência.

Risco 4 Dado limitações do *OpenZWave*, software escolhido para possibilitar a comunicação com dispositivos *ZWave*, pode haver incompatibilidades entre o dispositivo *ZWave* adquirido e o *OpenZWave*. No caso de incompatibilidade do dispositivo, este pode ficar parcial ou completamente inacessível pelo sistema, de modo que o investimento de recursos para adquirir o dispositivo foi não efetivo e este não poderá ser usado para o auxiliar no desenvolvimento do eaZy.

Mitigação: Adquirir dispositivos de Automação Residencial compatíveis com o sistema, levando em consideração recomendações de usuários e desenvolvedores de sistemas relacionados.

3.1.3 Requisitos

Para definir com mais clareza quais são as as necessidades do sistema eaZy, foram identificados os seus requisitos. Os requisitos foram separados entre Funcionais e Não-Funcionais, de acordo com [36]: Requisitos Funcionais são os que descrevem explicitamente as funcionalidades e serviços do sistema; enquanto os Requisitos Não-Funcionais são os que definem as propriedades e restrições do sistema.

Requisitos Funcionais

R.F. 1 Cada dispositivo físico da rede de dispositivos *ZWave* deve ser visualizado na lista de dispositivos do sistema.

R.F. 2 Para os dispositivos que suportam a troca de estado (*eg.* tomadas, lâmpadas), deve deve haver a possibilidade de troca de estados de um dispositivo.

R.F. 3 A lista de dispositivos deve conter as informações relevantes dos dispositivos (*eg.* consumo, nível de bateria, estado atual, tipo do dispositivo, nome do dispositivo e, caso disponível, dados de sensores).

R.F. 4 O sistema deve ser capaz de persistir as personalizações de usuário para cada dispositivo (*ie.* o nome do dispositivo dado pelo usuário).

R.F. 5 O usuário deve ser capaz de alterar as preferências dos dispositivos (*ie.* o nome do dispositivo dado pelo usuário).

R.F. 6 Cada dispositivo deve conter dois identificadores únicos: o identificador 'id', que identifica o dispositivo no dentro do banco de dados do sistema; e o identificador 'device_id', que identifica o dispositivo (nodo) na rede de dispositivos *ZWave*.

R.F. 7 Quando um dispositivo não está mais presente na rede de dispositivos *ZWave* e não é mais acessível pelo sistema, o usuário deve ser capaz de retirá-lo do banco de dados.

R.F. 8 No evento de atualização de dispositivo (*eg.* troca de estado, novo valor de sensor), a interface deve ser atualizada de acordo.

R.F. 9 O usuário deve ser capaz de definir um cenário simples: definir um estado que um dispositivo deve assumir, assim que algum dispositivo apresentar um atributo com valor escolhido pelo usuário.

R.F. 10 Os cenários devem ser armazenados no banco de dados, e conter identificadores 'id' (*ie.* identificador do dispositivo no banco de dados) para os dispositivos envolvidos nos cenários.

R.F. 11 O sistema deve checar cada troca de estado de dispositivo para verificar a existência de um cenário definido para aquele caso.

R.F. 12 O usuário deve poder remover cenários existentes, que são então excluídos do banco de dados.

Requisitos Não Funcionais

R.NF. 1 O sistema deve ser capaz de operar sobre um hardware de baixa performance e de custo reduzido (*ie.* Raspberry Pi).

R.NF. 2 Os dados de dispositivos visíveis na interface de usuário devem ser os mesmos presentes na Rede de Dispositivos *ZWave*, a qualquer momento.

R.NF. 3 A interface com usuário deve ser responsiva (*ie.* deve aderir ao dispositivo sendo usado pelo usuário naquele momento, como um dispositivo móvel ou tela do computador).

R.NF. 4 As diferentes telas da interface devem apresentar a mesma identidade visual.

R.NF. 5 O sistema deve ser compatível com Sistema Operacional Linux.

R.NF. 6 A arquitetura do sistema deve ser flexível, de modo que novos componentes possam ser criados e adicionados posteriormente com relativa facilidade, sem necessidade de mudanças estruturais ou grandes refatorações no sistema.

R.NF. 7 A licença de software do sistema deve ser de código aberto (*ie.* Licença MIT).

3.1.4 Cronograma

O Cronograma de Execução, que pode ser visualizado na Tabela 3.1, exibe a cronologia dos estudos que envolveram a elaboração do eaZy. A coluna 'Fase'do Cronograma de Execução compreende cada ítem presente na Sessão 1.5.2 Objetivos Específicos além das fases de redação e correção do presente documento.

Mês / Semana

	Abril				Maio				Junho				Julho				Agosto				Setembro				Outubro				Novembro			
Fase	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4
Estudo sobre Automação Residencial	■	■	■	■																												
Estudo sobre Tecnologias Relacionadas					■	■	■																									
Projeto e Arquitetura do eaZy											■	■	■	■																		
Implementação do eaZy														■	■	■	■	■	■	■	■	■										
Avaliação dos Resultados																							■	■								
Redação do Documento de tcc							■	■	■	■	■													■	■	■	■	■	■	■		
Correções do Documento de tcc																															■	■

Tabela 3.1: Cronograma de Execução

3.1.5 Arquitetura

A arquitetura do eaZy deve ser, de acordo com o requisito **R.NF. 6**, flexível e escalável. Senso assim, o sistema é dividido em três componentes, cada um com responsabilidade bem definida:

- **eaZy Web**: é o componente responsável pela **interface com o usuário**, que implementa uma interface *web* acessível por navegador de internet. Este componente apenas disponibiliza a interface em si, de modo que não guarda nenhum estado do sistema e nem persiste informações.

- **Home Stack**: é responsável pela lógica de Automação Residencial. Este é o **componente central** do sistema, que controla a persistência de dados, gerencia os cenários e provê toda a informação para o componente eaZy Web. Além disso, envia comandos para o Device Controller, e recebe dele as atualizações de dispositivos.

- **Device Controller**: componente responsável pela **comunicação com a rede de dispositivos *ZWave***, ou seja, é a interface entre o sistema e os dispositivos reais. O Device Controller não persiste dados sobre dispositivos. Sua função é repassar ao Home Stack as atualizações de dispositivos, e retornar as informações requeridas pelo Home Stack.

Os componentes que integram o eaZy funcionam de forma independente, e não existe uma ligação explícita entre eles. Cada componente é inicializado separadamente. Ainda assim, o eaZy funciona corretamente apenas quando todos os componentes estão devidamente inicializados.

Na Figura 3.2 pode-se ter uma visão geral de como o eaZy é arquitetado.

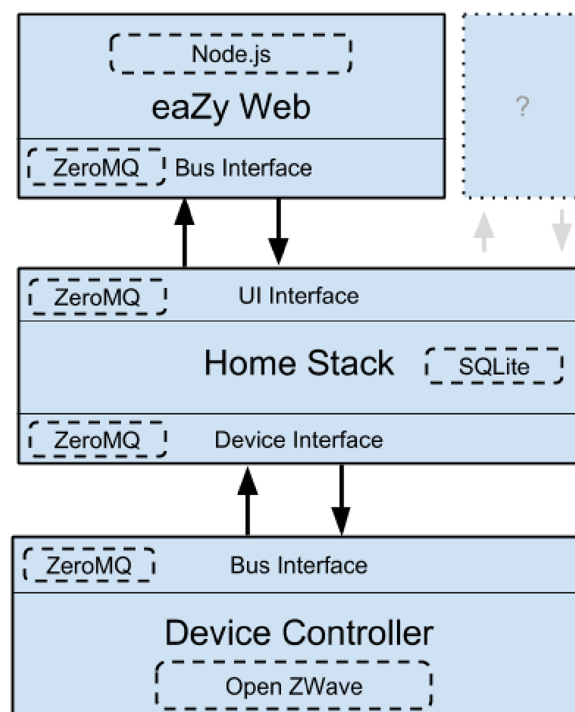


Figura 3.2: Arquitetura v3 do eaZy

Definição do meio de interação

Apesar da possibilidade de implementar mais de um componente de Interface com o Usuário, por conta da arquitetura flexível do eaZy, optou-se por implementar apenas uma interface de usuário nesta primeira versão do sistema. Dentre as opções consideradas citam-se Aplicativo para Dispositivo Móvel, Aplicativo Desktop e Aplicativo *Web*.

Escolheu-se então, por questão de flexibilidade e agilidade de desenvolvimento, pela implementação de uma interface *Web*, conforme documentado na sessão 3.4.4. Deste modo, apesar de implementar apenas um componente de interface, o utilizador possa acessar o eaZy de qualquer dispositivo que disponha de um navegador de internet.

Comunicação entre os componentes

A comunicação entre os componentes, representada por flechas na Figura 3.2, ocorre utilizando um padrão criado para o eaZy chamado de 'Comunicação por Barramento', explicado com mais detalhes na Sessão 3.4.1 Comunicação entre Componentes. A vantagem de utilizar a Comunicação por Barramento é o fato de que a arquitetura de comunicação do eaZy é uma mescla do modelo PubSub e Requisição-Resposta [37], possibilitando grande flexibilidade. No modelo PubSub, o Publicador (*ie.* componente Home Stack) pode distribuir mensagens, de modo que os Assinantes (*ie.* outros componentes) possam recebê-las. Além disso, os Assinante podem mandar mensagens direcionadas ao Publicador e, caso necessário, receber uma resposta direcionada, característica de um modelo Requisição-Resposta.

Com o modelo de Comunicação por Barramento, é possível que novos componentes sejam adicionados ao eaZy sem mudanças estruturais. Por exemplo, na Figura 3.2 há um pseudo-componente, representado com linhas pontilhadas. Este pseudo-componente poderia ser uma nova interface para o eaZy, por exemplo, uma aplicação para dispositivo móvel. Para que esta aplicação fosse acrescentada ao sistema, seria apenas necessário que esta implementasse o modelo de Comunicação por Barramento, então o componente Home Stack passaria a se comunicar com os dois componentes de interface gráfica sem sofrer modificações.

3.2 Subsídios Tecnológicos

Um fator chave para o desenvolvimento do eaZy foi a utilização de artefatos de software já existentes, de implementação de terceiros e licença livre, de modo que não haja um empreendimento de esforço desnecessário na construção de ferramentas.

Além disso, a utilização de Hardware embarcado é necessária para atingir um dos objetivos do eaZy, que é manter o custo final do sistema baixo e ser compacto em termos de tamanho.

Nesta sessão, são detalhados os artefatos de software de terceiros que foram incorporados ao sistema, além da descrição dos ítens de hardware utilizados durante a implementação e testes.

3.2.1 Software

Os principais artefatos de software de terceiros que compõem o eaZy são *OpenZWave* (comunicação com dispositivos), *Python OpenZWave* (biblioteca de abstração), *Zero MQ* (comunicação entre aplicações), *SQLite* (banco de dados), *Node.js* (*framework web back-end*) e *AngularJS* (*framework web front-end*). Abaixo segue uma descrição sobre cada artefato em si e como este é integrados ao eaZy, e em seguida um sumário com cada tecnologia e sua versão utilizada no sistema.

Open ZWave

O *Open ZWave* [17] é uma biblioteca de software livre (licença LGPLv3) que possibilita a comunicação entre o sistema que o utiliza, nesse caso o eaZy, e os dispositivos que implementam o ZWave [15], procolo de **comunicação com dispositivos** de Automação Residencial. Pelo fato do ZWave ser um protocolo proprietário da ZWave Alliance, sua especificação não é pública. Assim, a contrução da biblioteca *Open Zwave* ocorreu principalmente por engenharia reversa, observando o tráfego de dados entre os dispositivos que compõe a rede ZWave.

A desvantagem da biblioteca é o fato de não ser compatível com todos os dispositivos ZWave disponíveis no mercado, principalmente os relacionados à segurança (*eg.* fechaduras automatizadas, trancas de porta), por dificuldades em definir a especificação do protocolo de tais dispositivos.

Alternativas consideradas: Como alternativas para o (Open) ZWave, existem vários protocolos utilizadas para comunicação com dispositivos de Automação Residencial, que são detalhadamente estudadas na Sessão 2.3 Tecnologias de Comunicação entre Dispositivos de Automação Residencial, bem como os motivos para ter sido escolhido para integrar o eaZy na Sessão 2.3.1 Discussão sobre a tecnologia a ser utilizada.

Python Open ZWave

O *Python Open ZWave* [26] é uma **biblioteca de abstração do *Open ZWave*** para a linguagem de programação Python (v2.7). Pelo fato do Gerenciador de Dispositivos, componente do eaZy, ser implementado na linguagem Python, optou-se por utilizar a biblioteca de abstração ao invés de utilizar diretamente a implementação original do *Open ZWave* que é baseada na linguagem de programação C++. O *Python OpenZWave* é de licença livre, GPLv3.

A biblioteca permite basicamente todas as operações que o *Open ZWave* possibilita, mas com a vantagem de ter abstração para uma linguagem de alto nível que é o Python, de modo que a comunicação entre o eaZy e a rede de dispositivos ZWave possa acontecer com facilidade.

Alternativas consideradas: Existem outras bibliotecas de abstração do *Open ZWave* implementadas, como por exemplo o *Open ZWave Lua*, implementação de abstração para a linguagem de programação Lua. Contudo, escolheu-se pelo *Python Open ZWave* por que é compatível com as versões mais recentes do *Open ZWave* e por apresentar uma comunidade ativa de desenvolvedores e utilizadores.

Zero MQ

O *Zero MQ* [27] (ou ØMQ) é software de licença livre (LGPLv3) que possibilita a **comunicação entre aplicações**, sejam processos atuando em um único computador ou conectados via rede utilizando protocolo TCP. O *Zero MQ* disponibiliza uma biblioteca para troca de mensagens que abstrai a API de Sockets do sistema operacional, de modo que seja possível implementar, **com simplicidade**, padrões de comunicação entre processos como o *pub-sub* (onde processos 'assinantes', *sub*, recebem mensagens de um 'publicador', *pub*) ou *request-reply* (onde há a simples troca de mensagens entre dois processos).

A biblioteca pode ser utilizada em conjunto com grande parcela das linguagens de programação utilizadas atualmente [28], de modo que para cada linguagem suportada existem '*bindings*'que possibilitam que o *Zero MQ* seja incorporado ao software em questão.

O sistema eaZy é constituído por três componentes distintos: o Gerenciador de Dispositivos, o Home Stack e o eaZy Web. Estes três componentes são inicializados separadamente e operam de modo individual, sendo que a comunicação entre eles ocorre por meio da biblioteca *Zero MQ*. Como cada componente é implementado em uma linguagem de programação distinta, se fez necessário utilizar os *bindings* do *Zero MQ* para as linguagens Lua, Python e Javascript (Node.js).

Alternativas consideradas: Outros exemplos de bibliotecas de comunicação entre aplicações são o *Thrift* [31], da Apache, e o *Protocol Buffers* [32], da Google. Estas ferramentas exigem que o utilizador defina formatos padrões e utilize estruturas já definidas para implementar a comunicação, ao contrário do *Zero MQ* que possibilita que o usuário personalize sua arquitetura de comunicação. Por contar com mais simplicidade e flexibilidade, o *Zero MQ* foi escolhido para integrar o eaZy.

SQLite

O *SQLite* [33] é um **banco de dados** relacional, cujo código fonte é de domínio público. Sem necessita de um servidor, como no caso da maioria dos bancos de dados, o *SQLite* é integrado à aplicação que faz seu uso e não tem dependências de outras aplicações ou serviços. O foco deste banco de dados é a simplicidade e portabilidade, de modo que é ideal para uso em aplicações embarcadas.

Decorrente de sua simplicidade, o *SQLite* não tem suporte a algumas funcionalidades de gerenciamento de bancos de dados, como o gerenciamento de usuários do banco.

O requisito **R.F. 4** define que o componente Home Stack do eaZy persista informações do usuário e seus dispositivos, que deslancha na necessidade do uso de um banco de dados. O *SQLite* é usado no eaZy por conta de sua portabilidade, robustez e capacidade de operar com eficiência sob um hardware como o Raspberry Pi.

Alternativas consideradas: o *PostgreSQL* e o *MySQL* são bancos de dados de larga escala e foram considerados para integrarem o eaZy, mas por disponibilizarem funcionalidades que o sistema não necessita e dependerem de uma instância de servidor para sua execução, estes bancos de dados consumiriam recursos computacionais em excesso sem justificativa plausível.

Node.js

O *Node.js* [29] é um **framework para aplicações *web*** de código aberto (licença MIT) que é implementada utilizando o motor JavaScript V8, da empresa Google. A plataforma pode ser usada para construção de aplicações *web* de maneira rápida e simples, com suporte a operações de *streaming*, IO (operações envolvendo sistema de arquivos), *Web Sockets*, entre outros, por dispor de uma ampla variedade de bibliotecas convenientemente instaladas utilizando um gerenciador de pacotes próprio. A linguagem de programação do *framework* é JavaScript.

Por ser dotado de simplicidade, robustez e capacidade de operar sob o hardware Raspberry Pi, o *Node.js* foi escolhido para implementar a interface entre o usuário e o sistema eaZy, constituindo a parte *back-end*, ou seja, o servidor *web* do componente eaZy Web.

Alternativas consideradas: há muitos *frameworks* para aplicações *web* disponíveis, cada um com suas características próprias, como por exemplo: *Ruby on Rails* (Ruby), *Lapis* (Lua) e *Zend Framework* (PHP). Com a escolha de implementar o *back-end* em *Node.js*, há a vantagem de ter todo o componente eaZy Web implemewntado na linguagem de programação JavaScript.

AngularJS

O *Angular JS* [30] é um framework para desenvolvimento de aplicações *web* complexas que atua no *front-end* (licença MIT), ou seja, no navegador de internet do usuário. O framework, que é desenvolvido pela empresa Google, permite extensão da linguagem de marcação HTML (*HyperText Markup Language*), de modo que esta sirva de *template* para a aplicação, e utiliza a linguagem de programação JavaScript para implementação da lógica da aplicação.

Uma aplicação no *Angular JS* é implementada de modo modular, ou seja, é possível definir controladores logicamente de acordo com a interface e modelos da aplicação.

A interface de usuário do eaZy tem o requisito **R.F. 8**, especificando que a interface deve ser dinâmica e apresentar a possibilidade de atualizações e notificações de acordo com a mudança dos dispositivos do sistema. O *AngularJS* propõe um modelo estruturado para confecção de aplicações *web* capaz de suprir os requisitos de dinamismo do componente eaZy Web.

Alternativas consideradas: existem outros *frameworks* para desenvolvimento de aplicações *front-end* complexas, como por exemplo o *Ember.js* e o *CanJS*. A escolha do *Angular JS* deu-se por maior familiaridade do autor.

Sumário de Artefatos de Software

Segue abaixo, na Tabela 3.2, um sumário indicando os artefatos de software utilizados no desenvolvimento do eaZy com suas respectivas versões utilizadas.

Software	Descrição	Versão
Open ZWave	Comunicação com dispositivos ZWave	v1.0.791
Python Open ZWave	Biblioteca de abstração do Open ZWave	v0.2.6
Zero MQ	Comunicação entre aplicações	v3.2.4
SQLite	Banco de Dados	v3.7.9
Node.js	Framework *web front-end*	v0.10.26
AngularJS	Framework *web back-end*	v1.3.0

Tabela 3.2: Artefatos de Software

3.2.2 Hardware

O hardware escolhido para integrar o sistema proposto neste trabalho deve ser capaz de suportar a execução simultânea dos três componentes do eaZy, com eficiência, além de apresentar compatibilidade com os artefatos de Software escolhidos e com o Device Controller de Automação Residencial. Isso implica nos seguintes requisitos:

- apresentar arquitetura compatível com sistema o sistema operacional Linux;
- disponibilizar conexão Ethernet para comunicação com a rede de computadores; e
- compatibilidade com o Device Controller de Automação Residencial escolhido, por meio da conexão *Universal Serial Bus* (USB).

Raspberry Pi

O Raspberry Pi [39] é um computador de custo reduzido extremamente compacto, de tamanho similar ao de um cartão de crédito, produzido pela *Raspberry Pi Foundation*, uma fundação inglesa sem fins lucrativos. O Raspberry Pi utiliza processadores com arquitetura ARM, apresenta conexões USB, ethernet, HDMI e RCA. É disponibilizado em diferentes modelos, que variam em termos de especificação de hardware e preço. O Modelo B, utilizado pelo eaZy tem suas escificações detalhadas na Tabela 3.3 Especificação de Hardware do Raspberry Pi Modelo B.

Unidade de Processamento	700 MHz Low Power ARM 1176JZ-F
Memória	512MB SDRAM
Ethernet	OnBoard 10/100 Ethernet RJ45
Dimensões	8.6cm x 5.4cm x 1.7cm

Tabela 3.3: Especificação de Hardware do Raspberry Pi Modelo B.

O Raspberry Pi foi escolhido como hardware para servir de plataforma para o eaZy, levando em consideração o requisito **R.NF. 1**, por apresentar capacidade de processamento suficiente para suportar execução do sistema, ser compatível com sistema operacional Linux, ter conexão à rede de computadores

por meio de Ethernet e possuir conexão USB para utilização do Controlador de Dispositivos da rede de dispositovos de Automação Residencial. Além disso o Raspberry Pi, que pode ser visto na Figura 3.3 Raspberry Pi Modelo B, apresenta tamanho satisfatóriamente compacto.

Figura 3.3: Raspberry Pi Modelo B.
Fonte: http://www.raspberrypi.org/products/model-b/

O Sistema Operacional utilizado é o *Raspbian*, 'um sistema operacional baseado na distribuição Linux Debian e otimizado para o Raspberry Pi'[34], licença GPL. O *Raspbian* disponibiliza pacotes de configuração específicos para o Raspberry Pi, que auxiliam em configurações de vídeo e de rede, além de um grande número de pacotes de programas do *Debian* otimizados para a plataforma.

Dispositivos de Automação Residencial

Como descrito na Sessão 2.3.1 Discussão sobre a tecnologia a ser utilizada, o protocolo escolhido para integrar o eaZy foi o ZWave, por meio do software *Open ZWave* que possibilita a comunicação entre a rede de dispositivos e o sistema.

De acordo com o estudo da Sessão 3.2.1 Open ZWave, nem todos os dispositivos ZWave são compatíveis com o *Open Zwave*. Os dispositivos compatíveis foram identificados baseando-se em relatos da comunidade do *Open ZWave* [35], e dentre esses foram escolhidos os dispositivos a serem utilizados durante o desenvolvimento do eaZy.

Para que fosse possível obter uma simulação eficaz de um ambiente domiciliar automatizado, e ao mesmo tempo minimizar os custos de desenvolvimento do projeto, foram adquiridos apenas quatro dispositivos, que são um controlador, um sensor múltiplo e duas tomadas inteligentes:

- **Controlador**: O Device Controller é indispensável em uma rede de dispositivos ZWave, pois é o canal de comunicação entre os dispositivos da rede em questão e o hardware no qual reside o sistema de Automação Residencial. É por meio do Device Controller que o sistema reconhece a rede ZWave. Por exemplo: um sistema de Automação Residencial pode enviar comandos para o nodo Controlador, que então distribui o comando para os nodos destinatários, nos quais efetiva-se a ação desejada.

Existem vários modelos de Controladores ZWave de diferentes fabricantes. A métrica de escolha para o Device Controller que integra o eaZy é compatibilidade com o *Open ZWave*. O modelo escolhido, à partir de recomendações da comunidade de usuários e desenvolvedores do *Open ZWave* é o Aeotec Z-Stick S2, que pode ser visto na Figura 3.4.

Figura 3.4: Controlador ZWave: Aeotec Z-Stick S2.

- **Sensor Múltiplo**: Um sensor múltiplo é aquele capaz de obter mais de um tipo de informação sobre o ambiente no qual está inserido. No caso do Multisensor Homeseer HSM 100-S3, que pode ser visualizado na Figura 3.5, o Multisensor é capaz de mensurar temperatura e luminosidade, além de detectar presença. O dispositivo é compatível com *Open ZWave*.

  Utilizando este dispositivo durante o desenvolvimento do eaZy, foi possível adicionar à interface informações sobre temperatura e luminosidade do ambiente, além do estado dos dispositivos da rede. Também é possível alertar o usuário, caso o Multisensor detecta presença no ambiente.

Figura 3.5: Multisensor Homeseer HSM 100-S3.

- **Tomada Inteligente**: A tomada inteligente é aquela que pode ser controlada remotamente, além de obter informações sobre o aparelho que está ligado à ela, como consumo de energia elétrica. O modelo Aeon Labs Smart Energy Switch é uma tomada inteligente de simples utilização,

compatível com *Open ZWave*, pode ser visualizado na Figura 3.6.

Foram adquiridos dois modelos deste dispositivo, de maneira que durante o desenvolvimento do eaZy, foi possível controlar e obter informações sobre consumo de energia elétrica de dois aparelhos simultaneamente.

Figura 3.6: Aeon Labs DSC06106-ZWUS Smart Energy Switch.

Custo de Aquisição dos Artefatos de Hardware

Segue abaixo, na Tabela 3.4, o montante total de investimento, discriminado por ítem, dos artefatos de hardware adquiridos para o desenvolvimento do eaZy.

Ítem	**Quantidade**	**Custo (unidade)**
Raspberry Pi Modelo B	1	$ 46,00
Aeon DSA02203-ZWUS Labs Z-Wave Z-Stick 2	1	$ 42,75
Homeseer HSM 100-S3 Multi-Sensor	1	$ 97,23
Aeon Labs DSC06106-ZWUS Smart Energy Switch	2	$ 44,89
Total (dólares)		**$ 275,76**

Tabela 3.4: Custos de Aquisição dos Artefatos de Hardware

3.3 Processo de Desenvolvimento e Preparação

3.3.1 Dependências

Para que fosse possível desenvolver e testar os componentes do eaZy, foi primeiro necessário a instalação das dependências do sistema, ou seja, instalação dos artefatos de softwares de terceiros. Os artefatos

de terceiros utilizados estão descritos com detalhes na sessão 3.2.1 Software. Os *scripts* utilizados para instalação das dependências no Sistema Operacional Raspbian pode ser encontrado no Anexo A.1

3.3.2 Ferramental de Apoio

Durante toda a implementação do eaZy, seu código foi devidamente versionado utilizando a ferramenta Git [38]. Versionamento de Software é o ato de persistir as mudanças de um sistema de arquivos de acordo com o tempo, de modo que uma versão específica (ou mudança) pode ser localizada posteriormente e comparada à outras versões. É possível ver todo o histórico do desenvolvimento do sistema no repositório GitHub do autor, accessível pelo endereço eletrônico <`https://github.com/hoffmannmatheus/eaZy`>.

O sistema foi desenvolvido e testado em um computador utilizando Sistema Operacional Ubuntu 12.04 e, periodicamente, testado no Raspberry Pi, cujo Sistema Operacional é o Raspbian.

3.3.3 Desenvolvimento para Hardware Embarcado

Para que o sistema possa ser executado no Raspberry Pi, é necessário que todos os arquivos e dependências do eaZy sejam copiados para a plataforma alvo. Este processo é conhecido como *deploy*. Para que o *deploy* do eaZy seja feito com praticidade e sem consistência, foi implementado um *script* específico, que pode ser visto no Anexo A.3.

Para iniciar o sistema no Raspberry Pi, são utilizados dois *scripts*: o primeiro para configurar as variáveis de ambiente utilizadas pelo eaZy, encontrado no anexo A.2; e o segundo para efetivamente inicializar os componentes do sistema, encontrado no Anexo A.4. Além disso, foi criado um *trigger* para que os *scripts* de inicialização sejam executados automaticamente na inicialização do Raspberry Pi, de modo que o eaZy possa ser iniciado juntamente com a plataforma. A instalação do *trigger* pode ser visualizada no Anexo A.4.

3.4 Implementação

Nesta sessão é apresentado o processo de implementação do eaZy e como os componentes se comportam e interagem entre si. Estão descritas as integrações de artefatos de software de terceiros com o sistema e suas influências. Além disso, são descritas algumas das dificuldades enfrentadas durante a implementação, assim como as soluções para superá-las.

3.4.1 Comunicação entre Componentes

Para respeitar o requisito **R.NF. 6**, que define que o eaZy deve apresentar uma arquitetura flexível e escalável, foi desenvolvido um modelo próprio para a comunicação entre os componentes (ou processos) do sistema. Este modelo, chamado de **Comunicação por Barramento**, foi implementado de tal maneira que seja possível uma mescla de padrões existentes de comunicação entre processos: PubSub e Requisição-Resposta [37].

O modelo Comunicação por Barramento define duas entidades: o Cliente e o Servidor. Como pode ser visualizado na Figura 3.7, podem existir vários Clientes ligados à um único Servidor dentro do mesmo canal de comunicação. O canal de comunicação é baseado em Sockets (abstraídos utilizando a biblioteca *Zero MQ*) e é definido por três portas de Socket, utilizadas pelos Clientes e pelo Servidor deste canal.

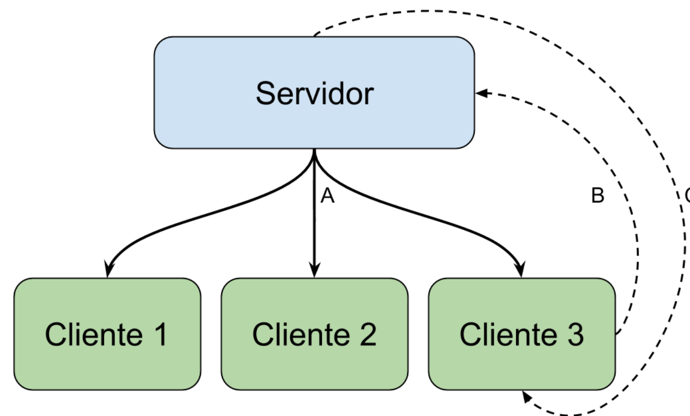


Figura 3.7: Modelo de Comunicação por Barramento.

Quando a entidade **Servidor** emite uma mensagem para o canal de comunicação, simbolizada pela letra A na Figura 3.7, todos os Clientes deste canal irão recebê-la (similar ao modelo PubSub). A entidade **Cliente** pode emitir uma mensagem, simbolizada pela letra B na Figura 3.7, que é então recebida apenas pelo Servidor, que, caso necessário, responde à esta mensagem diretamente para o Cliente remetente (similar ao modelo Requisição-Resposta), que é simbolizada pela letra C na Figura 3.7.

Para a implementação do modelo de Comunicação por Barramento, foi utilizada a biblioteca *Zero MQ*, que abstrai substancialmente a API de Sockets do sistema operacional. Além disso, pelo fato do Zero MQ possuir *bindings* para muitas linguagens, o modelo foi implementado em três linguagens: Lua (Home Stack), Python (Device Controller) e Javascript (eaZy Web). Foram implementadas bibliotecas separadas para as entidades Servidor e Cliente em Python e Lua. No caso do Javascript, o modelo foi integrado à aplicação, sem a produção de uma biblioteca à parte.

No eaZy existem dois canais de Comunicação por Barramento, como pode ser visto na Figura 3.2: entre os componentes Home Stack (Servidor) e eaZy Web (Cliente), e entre os componentes Home Stack (Servidor) e Device Controller (Cliente). A vantagem deste modelo é o fato que há **escalabilidade**: novos componentes podem ser criados e adicionados aos canais de comunicação existentes. Por exemplo: no caso da criação de um novo componente de Interface Gráfica para dispositivos móveis, para que o novo componente possa se comunicar com o Home Stack utilizando o canal de comunicação, este precisa apenas integrar a entidade Cliente do modelo, de modo que este já passaria a receber e enviar mensagens utilizando o canal de comunicação. Além disso, há **flexibilidade**: no mesmo componente, o Home Stack apresenta dois canais de comunicação independentes. Um canal para a comunicação com componentes de interface gráfica, e outro canal para os componentes controladores de dispositivos.

3.4.2 Componente Home Stack

O Home Stack é o componente central do sistema, uma vez que este é responsável pela lógica de Automação Residencial do eaZy. Entre as atribuições do componente, estão:

- Validar ações referentes a dispositivos de Automação Residencial e enviá-los para os componentes de controle de dispositivos (*ie.* Device Controller).
- Responder às requisições dos componentes de interface (*ie.* eaZy Web).
- Persistir as preferências de usuário em banco de dados (requisito **R.F. 4**).
- Persistir os cenários de automação definidos pelo usuário (requisito **R.F. 10**).
- Checar cada atualização da rede de dispositivos para verificar possíveis cenários e ativá-los caso necessário (requisito **R.F. 11**).

Escolheu-se implementar o componente na linguagem de programação Lua, que é uma linguagem interpretada que provê flexibilidade e funcionalidades simples e robustas, por meio de uma sintaxe expressiva. O Home Stack deve ter interface com banco de dados e integrar o modelo de Comunicação por Barramento, então é necessária a instalação de pacotes externos para o Lua, que é possível ao utilizar um gerenciador de pacotes chamado de *luarocks*. No caso do *Home Stack*, foram instalados os pacotes *lsqlite3* (interface para o banco de dados *SQLite*) e *lzmq* (interface com o *Zero MQ*).

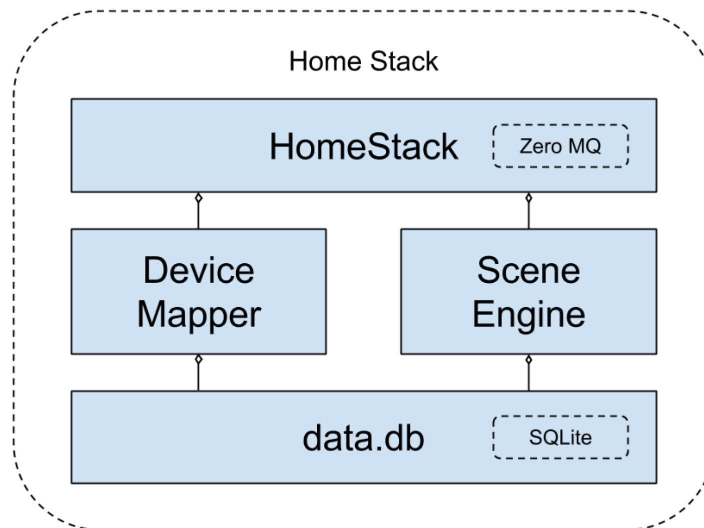


Figura 3.8: Estrutura do componente Home Stack.

A estrutura do Home Stack, que é representada pela Figura 3.8, é segmentada em quatro partes:

- O módulo **HomeStack**, que é o módulo principal. É responsável pela comunicação entre este componente e os demais, e distribuir a lógica de cada evento para o módulo correto. Este módulo utiliza a biblioteca de Comunicação por Barramento em *Lua*, apresentada na sessão 3.4.1, para comunicação entre componentes por meio do *Zero MQ*.

- O módulo **DeviceMapper**, que é responsável pelo mapeamento entre o dispositivo real, presente na rede de dispositivos, e o respectivo dispositivo persistido em banco de dados. O mapeamento é necessário para que as personalizações do usuário sejam atribuídas ao dispositivo correto, de modo que cada atualização originada do componente Device Controller é direcionada ao eaZy Web com todas as informações do dispositivo. É responsável também por reconhecer se um determinado dispositivo foi removido ou adicionado à rede de dispositivos.

- O módulo **SceneEngine**, que é responsável pelo gerenciamento dos cenários criados pelo usuário. Este módulo efetua a checagem de possíveis cenários para cada atualização da rede de dispositivos, além de permitir a criação e remoção dos cenários.

- O módulo **data.db**, que é reponsável pelo gerenciamento de banco de dados para o componente Home Stack. Este módulo utiliza o pacote *lsqlite3* para efetuar seleção, remoção, atualização e inserção dos dados do eaZy ao banco de dados *SQLite*.

3.4.3 Componente Device Controller

O componente Device Controller é responsável pela comunicação entre o eaZy e a rede de dispositivos *ZWave*, de modo que:

- Constrói a lista de dispositivos a ser enviada para o Home Stack (requisito **R.F. 1**).

- Recebe comandos do componente Home Stack e os traduz para eventos equivalentes, enviando-os para a rede de dispositivos (requisito **R.F. 2**).

- Recebe atualizações da rede de dispositivos *ZWave*, filtra as atualizações pertinentes e as envia para o Home Stack (requisito **R.F. 8**).

O componente Device Controller é implementado utilizando a linguagem de programação *Python*, uma linguagem interpretada que apresenta muitas funcionalidades, performance e robustez. Além disso, a implementação em *Python* faz com que a integração com a biblioteca *Python OpenZWave* (descrita na sessão 3.2.1) seja trivial, pelo fato desta ser implementada na mesma linguagem.

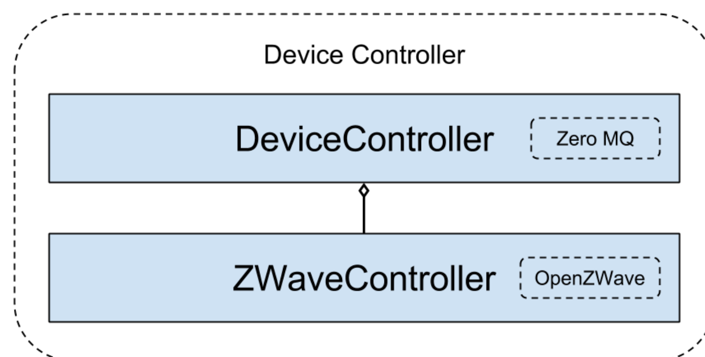


Figura 3.9: Estrutura do componente Device Controller.

Como visto na Figura 3.9, o componente é segmentado em duas partes:

- O **DeviceController**, módulo principal do componente Device Controller. Este módulo se comunica com o componente Home Stack utilizando a biblioteca de Comunicação por Barramento em *Python*, apresentada na sessão 3.4.1, para comunicação entre componentes por meio do *Zero MQ*. O módulo processa as solicitações recebidas e distribui a lógica para o ZWaveController, além de enviar as atualizações apontadas pelo módulo ZWaveControler para o componente Home Stack.

- O módulo **ZWaveController** é responsável pela interface com a rede de dispositivos *ZWave*, e utiliza a biblioteca *Python OpenZWave* para receber informações sobre a rede e enviar comandos para ela. Este módulo traduz a lista de nodos da rede de dispositivos *ZWave* para uma lista de dispositivos do eaZy. Além disso, recebe atualizações da rede, as filtra e enviar as atualizações pertinentes ao módulo DeviceController.

As atualizações enviadas da bibliotecas *OpenZWave* para o componente Device Controller são filtradas para que apenas as atualizações pertinentes sejam efetivamente enviadas. Como há muitos atributos associados a cada dispositivo da rede, muitas atualizações podem não ser pertinentes porque não influenciam no sistema como um todo. As atualizações enviadas do Device Controller para o Home Stack somente tangem os atributos de dispositivos que são utilizados pelo eaZy, como por exemplo: consumo de energia elétrica, nível de bateria, dados de sensores e estado do dispositivos.

3.4.4 Componente de Interface com Usuário – eaZy Web

A interface de usuário do eaZy é de responsabilidade do componente eaZy Web. O componente provê interação entre o usuário e o sistema, de modo que suas atribuições são:

- Disponibilizar uma Interface de Usuário para o sistema eaZy, de modo que este possa ter acesso às suas funcionalidades com praticidade. A interface deve ser responsiva a apresentar consistência entre as diversas telas (requisitos **R.F. 3**, **R.NF. 3** e **R.NF. 4**).

- Requisitar dados do eaZy (*ie.* lista de dispositivos e a lista de cenários) para o componente Home Stack, sem que o componente eaZy Web persista dados.

- Possibilidade de alterar estado de um determinado componente, como por exemplo acender uma lâmpada (requisito **R.F. 2**)

- Prover meios para personalização de dispositivos, além da criação, listagem e remoção dos cenários (requisitos **R.F. 5** e **R.F. 9**).

- Estar disponível para receber atualizações de dispositivos, originadas do componente Home Stack, e atualizar a Interface do Usuário com os novos dados (requisito **R.F. 8**).

Na escolha do meio de interação com usuário, como visto na sessão 3.1.5, definiu-se pela interface *web*. A tecnologias utilizadas para implementação do componente foram definidas na sessão 3.2.1: *Node.js* (como servidor *web*) e *AngularJS* (para lógica de renderização de conteúdo para o cliente). A Figura 3.10 demonstra a estruturação do componente eaZy Web.

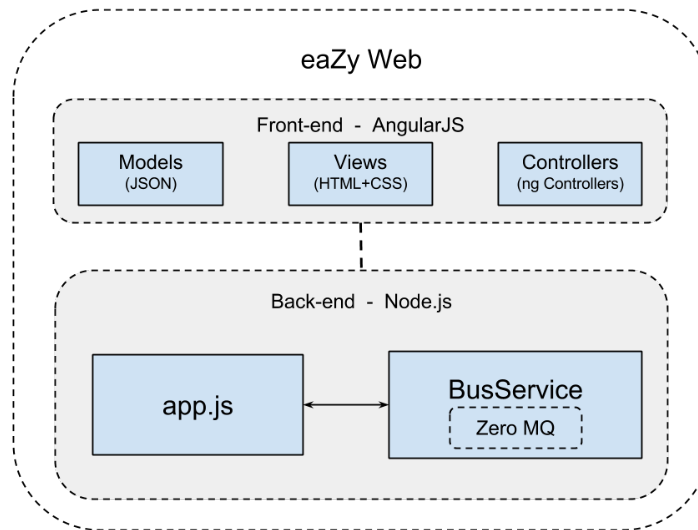


Figura 3.10: Estrutura do componente eaZy Web.

A estrutura do eaZy Web é dividida entre *back-end* e *front-end*. Ambos os segmentos são implementados utilizando a linguagem de programação Javascript. Os segmentos:

- **Servidor - *Back-End***: O módulo *back-end* do eaZy Web é responsável por servir as páginas de internet para os clientes, que utilizam sua API RESTful para comunicação. É implementado utilizando o *Node.js*. Este módulo também possibilita a comunicação entre os componentes eaZy Web e Home Stack por meio do serviço chamado de **BusService**. O BusService faz uso do *Zero MQ* para possibilitar ao componente o acesso ao modelo de Comunicação por Barramento.

- **Páginas Web - *Front-End***: Módulo responsável pela apresentação da Interface de Usuário. Além de *HTML+CSS* para a estruturação das páginas, o *framework AngularJS* é utilizado para adicionar lógica ao *Front-End* do eaZy Web. O módulo utiliza *web-sockets* para receber atualizações do servidor, de modo que as notificações recebidas são tratadas e a interface é atualizada quando necessário. A interface é construída com uso de bibliotecas (*eg. bootstrap*) que auxiliam na produção de páginas responsivas e consistentes entre si.

Fluxo da Interface de Usuário

Para que a interação entre o usuário e o sistema seja gratificante e exija o mínimo de esforço possível, foi definido um fluxo da interface de modo que as funcionalidades do eaZy sejam acessíveis facilmente.

Foi definido que haverá uma página para cada funcionalidade do sistema, que serão acessíveis a partir da página inicial chamada de emphdashboard:

- *Dashboard*: página inicial do sistema, que contém os *links* para as outras páginas, *Devices*, *Scenes* e *Energy*. Além disso, a página mostra um resumo de informações sobre os dispositivos do eaZy.

- *Devices*: esta página contém a lista de dispositivos do sistema e possibilita interagir com cada dispositivo.

- *Scenes*: página de listagem e criação de cenários do eaZy.
- *Energy*: esta página demonstra informações sobre consumo de energia elétrica sobre os dispositivos do sistema.

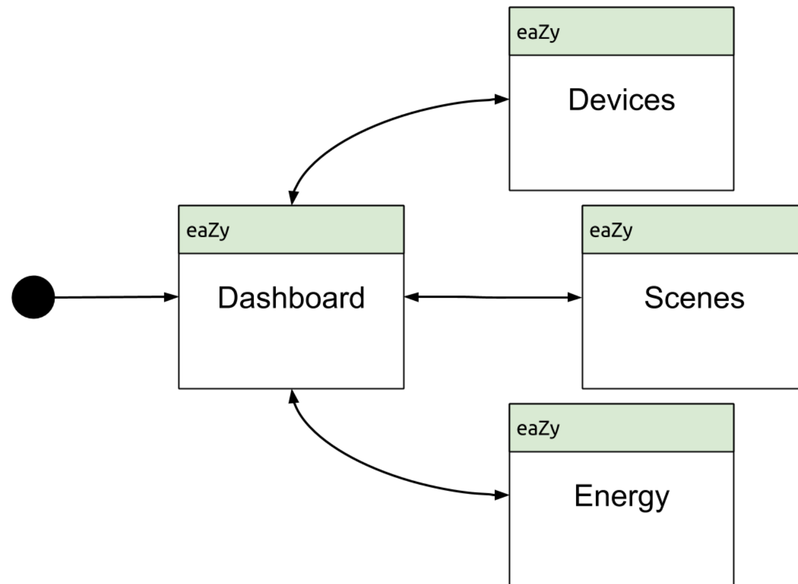


Figura 3.11: Diagrama de fluxo da Interface de Usuário do eaZy.

3.5 Interação entre componentes

Uma vez descrito cada componente do eaZy separadamente, é interessante uma visão do funcionamento do sistema como um todo. Nesta sessão são apresentados dois segmentos importantes do sistema, no formato de diagramas.

Estados dos Dispositivos

Do ponto de vista da rede física, um dispositivo pode estar acessível ou não. Já do ponto de vista do sistema, um dispositivo pode estar registrado, não registrado ou ausente. O diagrama abaixo demonstra os possíveis estados de um dispositivo no eaZy, representado no Diagrama de Estados da Figura 3.12.

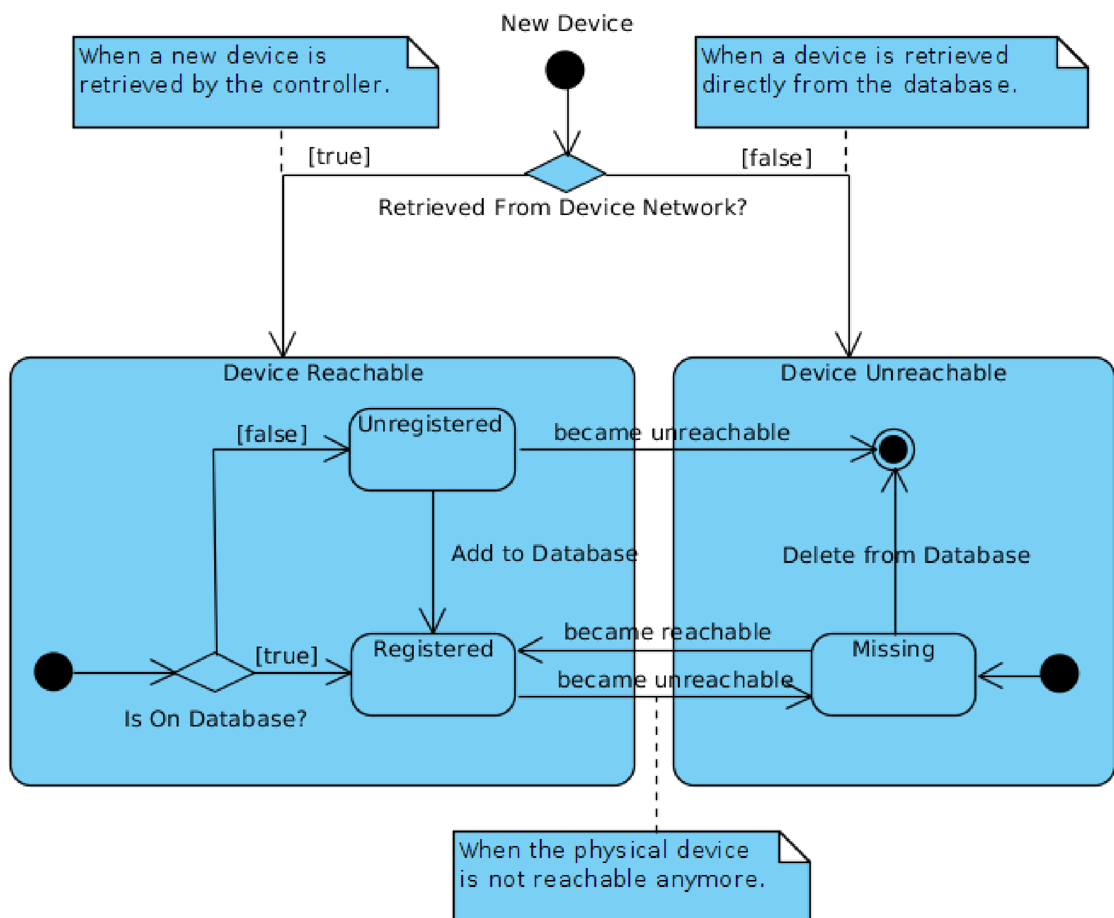


Figura 3.12: Diagrama de Estados de um dispositivo no sistema.

Listagem de Dispositivos

O Diagrama de Sequência para a funcionalidade de Listagem de Dispositivos, que pode ser visualizado na Figura 3.13, demonstra o comportamento do sistema no momento em que o usuário requisita a lista de dispositivos (*ie.* acessa a interface *web* do sistema). Esta funcionalidade abrange todos os componentes do sistema, então é possível visualizar a interação entre os componentes por meio da Comunicação por Barramento.

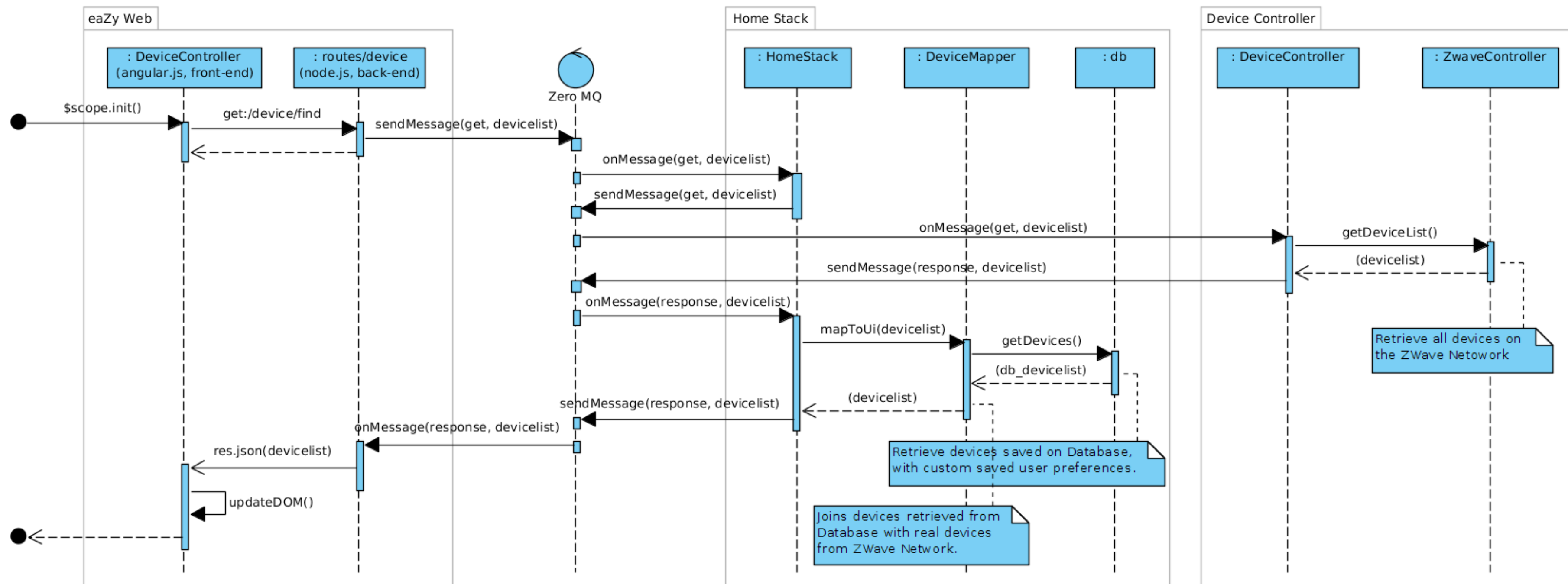


Figura 3.13: Diagrama de Sequência da funcionalidade de listagem de dispositivos.

Capítulo 4

Resultados Obtidos

Neste capítulo são expostos os resultados do trabalho, ou seja, o software desenvolvido no Capítulo 3 Metodologia. Primeiramente, é demonstrado o conjunto de Dispositivos de Automação Residencial utilizados neste capítulo. Em seguida, demonstra-se a Interface de Usuário do eaZy por meio de capturas de tela. Finalmente, são expostos dados no formato gráfico referentes ao desenvolvimento do sistema.

4.1 Código Fonte

Como produto principal deste trabalho, apresenta-se o código fonte do eaZy. O código pode ser visualizado por meio da ferramenta *Github*, onde encontra-se hospedado o repositório do projeto no endereço eletrônico <`https://github.com/hoffmannmatheus/eaZy`>, ou ainda no Anexo B deste documento.

4.2 Dispositivos de Automação Residencial Utilizados

Como descrito na Sessão 3.2.2 Hardware do Capítulo de Metodologia, foram adquiridos Dispositivos de Automação Residencial para que fosse possível o desenvolvimento e testes do sistema proposto. Os dispositivos utilizados são os da Figura 4.1, e são, respectivamente da esquerda para a direita: Raspberry Pi (Modelo B), Controlador ZWave (*Aeotec Z-Stick S2*), Sensor Múltiplo ZWave (*Homeseer HSM 100-S3*) e duas Tomadas Inteligentes ZWave (*Aeon Labs Smart Energy Switch*).

4.3 Demonstração do Sistema

Nesta sessão, o sistema é demonstrado por meio de capturas de tela da Interface de Usuário, como descrita na sessão 3.4.4 Fluxo da Interface de Usuário. A interface de usuário é disponibiliza pelo componente eaZy Web, acessível por meio de navegador de internet. Para demonstrar a responsividade da interface, são demonstradas capturas de tela de computador (Sistema Operacional Ubuntu) e também de dispositivo móvel (Sistema Operacional Android), ambos conectados à mesma rede de computadores à qual o Raspberry Pi eaZy opera.

Figura 4.1: Dispositivos utilizados para o desenvolvimento do eaZy

Os dispositivos do sistema podem receber nomes personalizados pelo usuário (o nome padrão do dispositivo é o nome de fábrica). Nesta sessão, as capturas de tela demonstram os dispositivos utilizados na Figura 4.1, que receberam nomes personalizados: o dispositivo Multisensor ZWave tem o nome 'Room Multisensor'; a Tomada Inteligente 1 tem o nome 'PC Monitor'; e a Tomada Inteligente 2 tem o nome 'Room Air Conditioner'.

Página inicial – *Dashboard*

Na Figura 4.2 pode-se visualizar a página inicial do eaZy capturada à partir de um Desktop. O *Dashboard* exibe uma visão geral do sistema, como o total de consumo de energia no momento, temperatura e a quantidade de dispositivos ligados. Além disso, a página contém ligações para as outras páginas da interface (a página *Settings* não é acessível pois sua implementação não é prevista para esta primeira versão do eaZy).

O *dashboard* apresenta um formato diferente ao ser acessado à partir de um dispositivo móvel, de modo responsivo. As Figura 4.3 demonstra a página como visualizada de um dispositivo móvel.

Lista de Dispositivos – *Devices*

Na página *Devices* é possível ter acesso à lista de dispositivos do sistema. Cada dispositivo listado apresenta informações sobre o estado deste dispositivo e seus atributos, como pode ser visto na Figura 4.4.

À partir desta lista é possível alter os estados dos dispositivos que suportam esta funcionalidade, como os dispositivos 'Room Air Conditioner'e 'PC Monitor', clicando na área do dispositivo em questão.

Além disso, é possível personalizar o dispositivo (*ie.* alterar o seu nome). Esta funcionalidade é acessível, no caso da interface via Desktop, ao clicar no símbolo do lado direito da área do dispositivo.

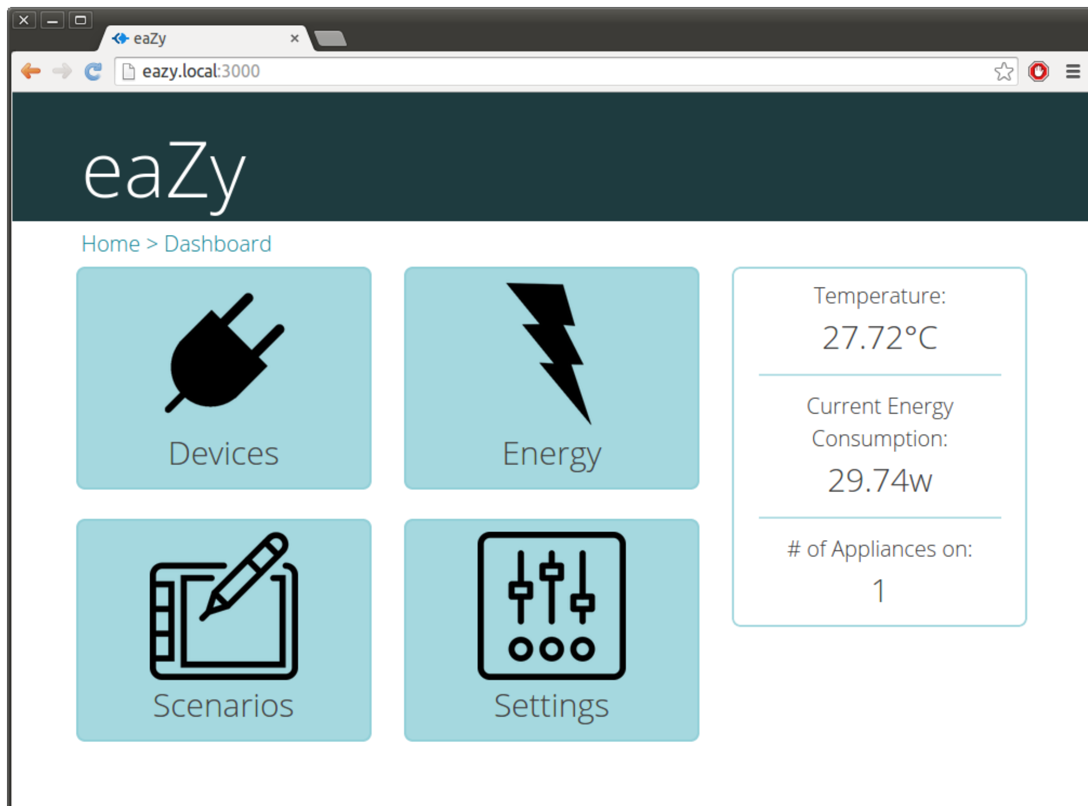
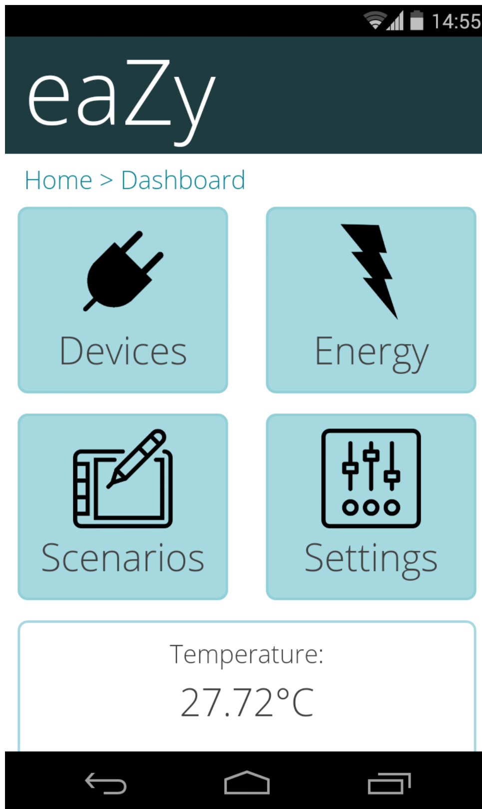
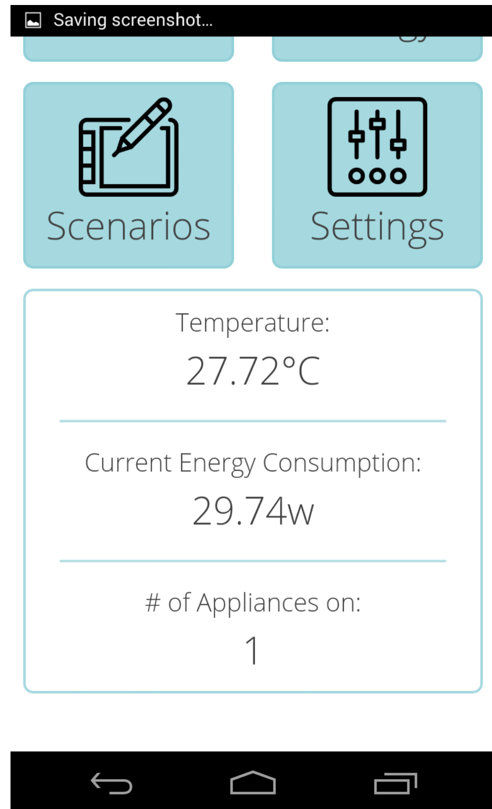


Figura 4.2: Captura de Tela Desktop do *Dashboard* do eaZy.

No caso da interface à partir de dispositivo móvel, como pode ser visualizado na Figura 4.5, o ícone não existe. O usuário deve utilizar um gesto na tela do dispositivo para habilitar a área de edição do dispositivo.



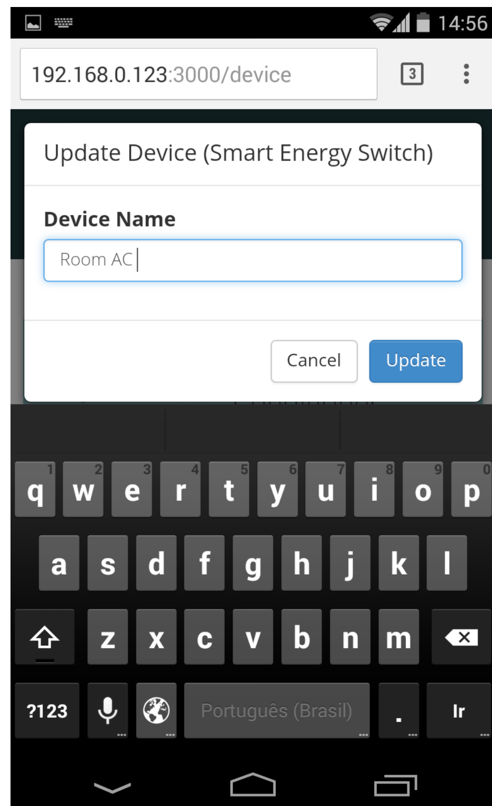
(a) *Dashboard* do eaZy parte 1.



(b) *Dashboard* do eaZy parte 2.

Figura 4.3: Captura de Tela Dispositivo Móvel do *Dashboard* do eaZy.

(a) Lista de dispositivos do eaZy.



(b) Edição do nome do dispositivo.

Figura 4.5: Captura de Tela Dispositivo Móvel da Lista de Dispositivos eaZy.

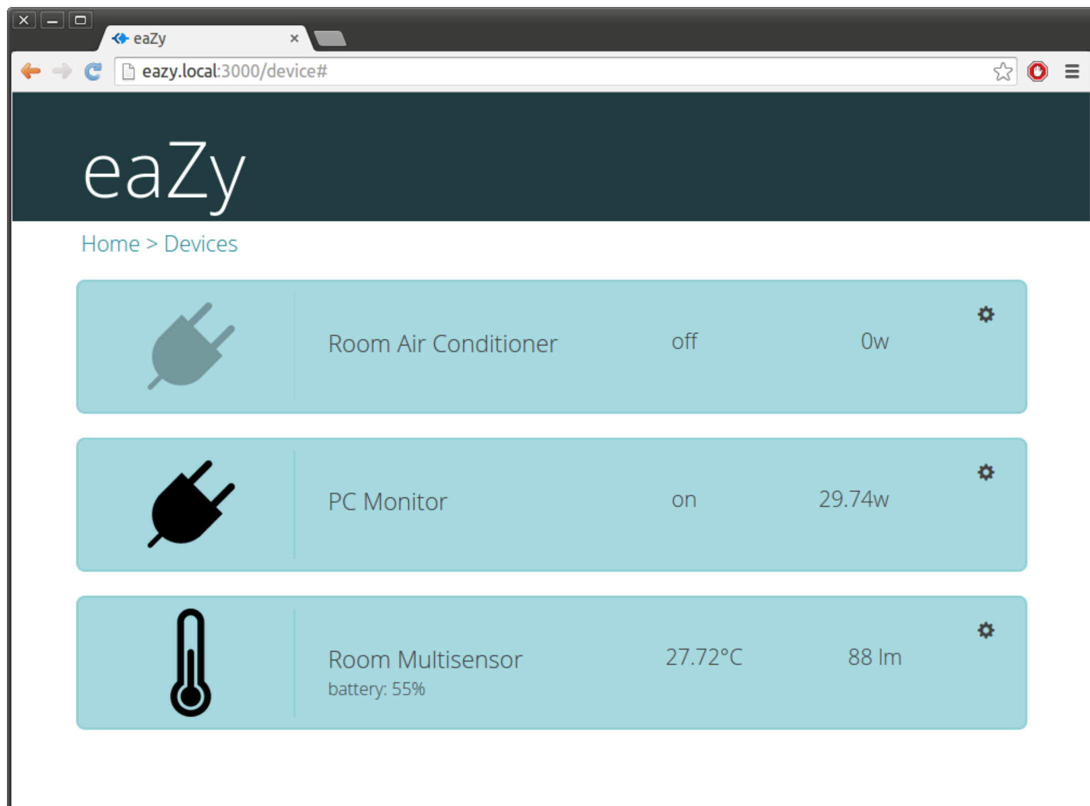


Figura 4.4: Captura de Tela Desktop da Lista de Dispositivos do EAZY.

Gerenciador de Cenários – *Scenes*

Na página do Gerenciador de Cenários, o usuário pode visualizar os cenários já criados, excluir um cenário ou ainda criar novos cenários. Como pode ser visualizado na Figura 4.6, as cenas são expostas no formato de tabela, determinando que quando um determinado dispositivo atingir um estado, o outro dispositivo em questão assumirá o estado novo.

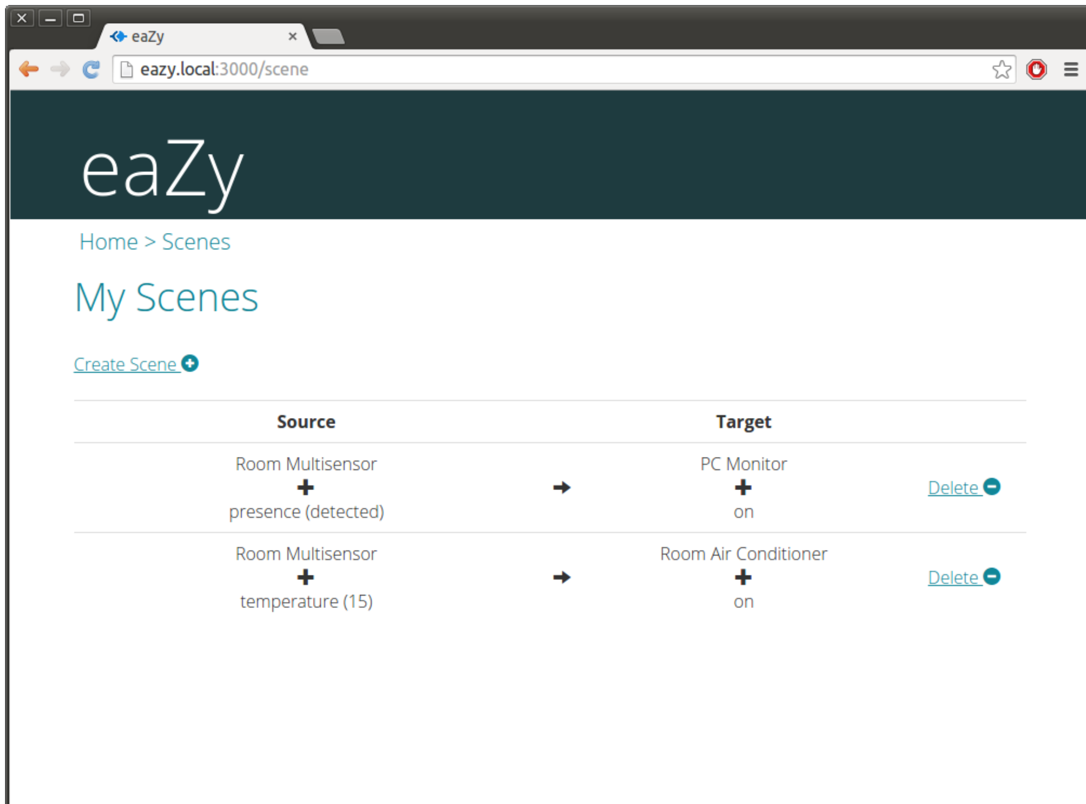


Figura 4.6: Captura de Tela Desktop do Gerenciador de Cenários do eaZy.

Para a criação de cenários foi desenvolvido um formulário, demonstrado na Figura 4.7. Ao preencher o formulário, o usuário pode selecionar um dispositivo da lista para ser o dispositivo 'fonte' do evento. Então, na próxima etapa de seleção de atributos, a lista é preenchida com os atributos do dispositivo 'fonte'. O usuário então preenche um valor desejado, seleciona o dispositivo 'alvo', e o novo estado do dispositivo 'alvo'.

Segue a descrição verbal do cenário de exemplo da Figura 4.7: quando o atributo *temperatura* do dispositivo *Room Multisensor* registrar um valor igual à *28*, o dispositivo *Room Air Conditioner* será ligado.

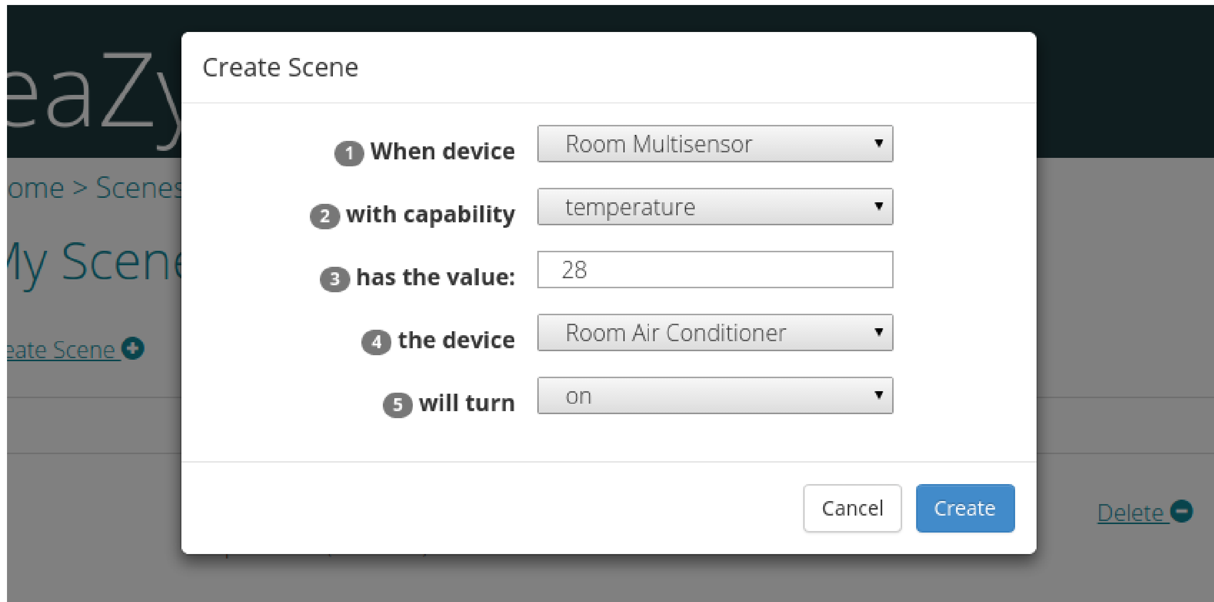


Figura 4.7: Captura de Tela Desktop do Formulário de Criação de um novo Cenário do eaZy.

A apresentação da página de Gerenciamento de Cenários é similar no dispositivo móvel, como pode ser visualizado na Figura 4.8.

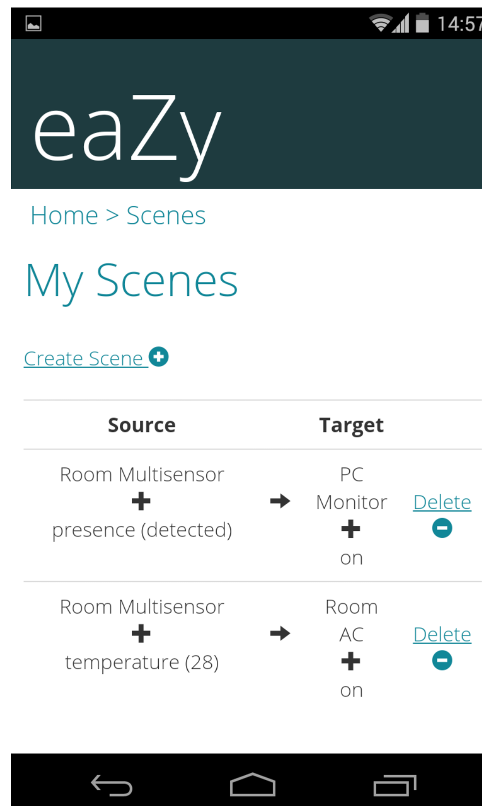


Figura 4.8: Captura de Tela Dispositivo Móvel do Gerenciador de Cenário do eaZy.

Página de Informações sobre Consumo – *Energy*

Na página de Informações sobre Consumo, é possível ver os valores de consumo de energia elétrica total dos dispositivos ligados ao eaZy. Há o valor total, somatório de todos os dispositivos, e também o valor por dispositivo, no formato tabular. A página pode ser visualizada na Figura 4.9.

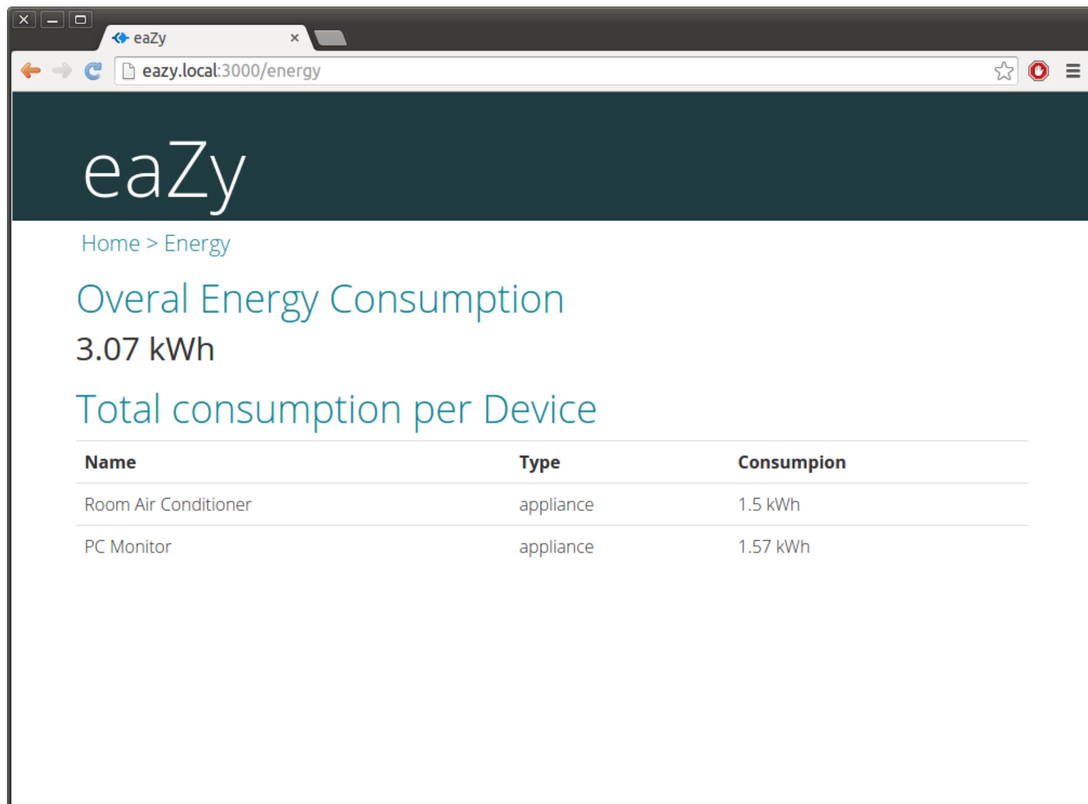


Figura 4.9: Captura de Tela Desktop de Informações sobre Consumo do eaZy.

4.4 Dados sobre o Desenvolvimento do Sistema

Para demonstrar a progressão do desenvolvimento do sistema eaZy, segue abaixo, na Figura 4.10, um gráfico referente ao número de *commits* por semana, entre os períodos de junho/2014 até outubro/2014.

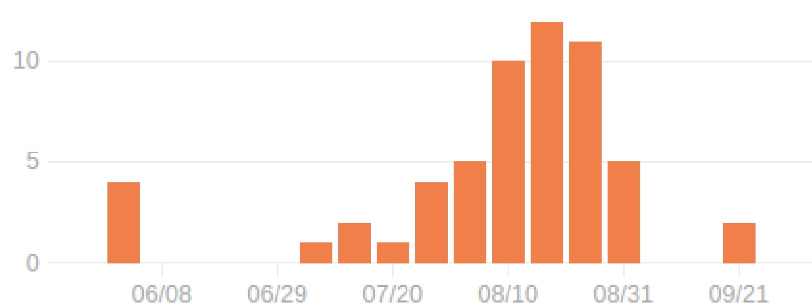


Figura 4.10: Gráfico de *commits* por semana do desenvolvimento do eaZy.

Commit é a denominação para uma nova versão do projeto, que contem modificação em relação

à última versão. Normalmente um novo *commit* é feito a cada implementação de uma nova *feature* ou progresso significativo. No caso do eaZy pode-se ver que o ponto auge de desenvolvimento foi na segunda semana de agosto, no qual foram realizados doze *commits*.

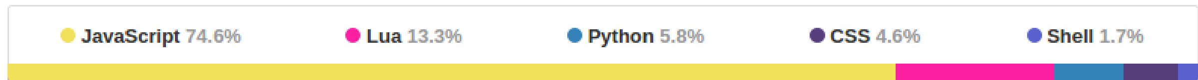


Figura 4.11: Linguagens de programação utilizadas no eaZy.

Os gráficos demonstrados nesta sessão foram extraídos da ferramenta *Github*, que serviu de hospedagem para o repositório de versionamento de código *Git* do projeto, como descrito na Sessão 3.3.2.

Capítulo 5

Conclusões

Este é o capítulo de fechamento do trabalho, onde são discutidos os Resultados obtidos, as decisões de implementação, as dificuldades superadas durante o desenvolvimento e projeto, e ainda são sugeridos alguns trabalhos futuros.

5.1 Discussão

Nesta sessão são discutidas as dificuldades de projeto e implementação do eaZy e como estas dificuldades foram superadas, além da avaliação final da solução.

5.1.1 Dificuldades e soluções

Durante a concepção do eaZy surgiram uma série de dificuldades relacionadas ao projeto e desenvolvimento do software. Seguem abaixo algumas das dificuldades mais impactantes e soluções adotadas.

- Projeto flexível: um dos principais requisitos do eaZy é a flexibilidade do sistema. Segundo o requisito **R.NF. 6**, o sistema deve ser flexível e prover baixo acoplamento das tecnologias de Automação Residencial e de Interação com Usuário, sem que a lógica do sistema seja comprometida.
  A solução encontrada foi desenvolver um padrão de Comunicação por Barramento (descrito na sessão 3.4.1) que permite uma arquitetura mais flexível, de modo que com componentes possam comunicar-se entre si independentemente da tecnologia que implementa cada componente.

- Atualização da Interface: segundo o requisito **R.F. 8**, o a interface deve ser dinâmica e atualizar à medida os dispositivos e sensores reportem novos dados. Durante o desenvolvimento do eaZy Web foi primeiramente utilizado o *framework Sails.js*, que apesar de facilitar implementação de rotas, controles e políticas de acesso ao servidor web, trouxe dificuldades para estabelecer a comuniação via *Web Sockets* entre o navegador e o servidor. Esta comunicação por *Web Sockets* é importante, pois possibilita um meio prático para enviar atualizações broadcast para todos os clientes (navegadores de internet) conectados ao sistema.
  Para solucionar o problema, o *framework Sails.js* foi removido para dar lugar apenas a tecnologia

Node.js, juntamente com o pacote *Socket.io*. Assim, o servidor pode enviar mensagens facilmente para os clientes.

- Falta de documentação da API *Python Open ZWave*: durante o desenvolvimento do Device Controller foi utilizado do *framework Python Open ZWave*, para comunicação entre o componente e os dispositivos de Automação Residencial. A dificuldade fez-se presente no momento de utilizar a API do *framework*, pois esta não apresenta uma documentação ampla sobre sua utilização. Para que fosse possível a implementação do Device Controller, foi necessário descobrir empiricamente, por método de tentativa e erro, sobre como algumas funcionalidades do *framework* devem ser utilizadas e então implementar a comunicação entre o sistema e os dispositivos de Automação Residencial.

- Definição de padrão para comunicação: ao utilizar o padrão de Comunicação por Barramento, a comunicação entre os componentes ocorre por troca de mensagens em formato de texto. Não sendo possível enviar objetos entre um componente e outro, principalmente pelo fato de que cada componente é implementado numa linguagem diferente, foi necessário definir um padrão para as mensagens. Foi definido que as mensagens a trafegarem pelo modelo de Comunicação por Barramento devem ser no formato JSON (*JavaScript Object Notation*), que contenha no mínimo três atributos: 'sender', para identificar o remetente; 'type', para identificar o tipo da mensagem, e; 'data', que é o conteúdo da mensagem. Abaixo segue um exemplo de mensagem utilizada:

```
{"sender":"home_stack","data":"devicelist","type":"get"}
```

5.1.2 Avaliação da Solução

Como resultado do esforço de desenvolvimento do projeto, obtém-se a solução final tal qual idealizada. O eaZy cumpre os requisitos estipulados e consegue atingir um custo final interessante para uma solução de Automação Residencial. O custo final do sistema, melhor discriminado na Sessão 3.2.2 Custo de Aquisição dos Artefatos de Hardware, foi de $ 275,76.

No geral, o sistema é robusto e responsivo, de modo que consegue manter-se atualizado de acordo com os dispositivos reais e interagir com estes eficientemente.

O sistema é funcional, mas pode receber algumas melhorias. Um exemplo é a interface do eaZy Web visualizada em um dispositivo móvel: alguns elementos podem ser posicionados e dimensionados para melhor acomodar-se à interface. Além disso, há espaço para melhorias de usabilidade, como por exemplo gestos mais intuitivos para edição dos dispositivos na lista.

5.2 Trabalhos Futuros

Como Trabalhos Futuros, são identificadas alguns aprimoramentos e novas funcionalidades que podem ser incorporadas ao sistema:

- **Medidas de Segurança**: adicionar controle de acesso à Interface de Usuário, com autenticação de sessão para cada requisição à API REST do componente eaZy Web.

- Aprimoramentos no componente de **Interface de Usuário**:
  - ajustar detalhes para tornar a apresntação mais atrativa e adicionar mais responsividade à Interface de Usuário, de modo que todo o conteúdo apresentado pelo componente eaZy Web seja coerente independente da interface sendo utilizada, dispositivo móvel ou computador.
  - O componente eaZy Web poderia contar com um sistema de registro de atividades do usuário, para que as ações do usuário possam ser persistidas e seja possível a visualização do histórico de atividades.
- Aprimoramentos para o gerenciador de cenas, **Scene Engine**, do componente Home Stack:
  - Opção de para impedir que a cena seja executada mais de uma vez, em um determinado espaço de tempo.
  - Opção de inverter o estado do dispositivo alvo da cena, de modo que a mesma ação pode gerar resultados diferentes, de acordo com o estado atual do dispositivo.
  - A cena não deve ser executada desnecessariamente: se o estado alvo da cena já seja o estado em vigor do sistema, a execução é adiada.
- Implementação de testes de software, com foco nas funcionalidades chave do sistema, que envolvem o gerenciamento de cenas e controle de dispositivos.
- Novo **Componente de Estatísticas**: Implementar um novo componente para geração de estatísticas. Este novo componente pode utilizar o modelo de Comunicação por Barramento para persistir toda a atividade do sistema. Estes dados podem servir de subsídio para gerar estatísticas de consumo de energia elétrica, hábitos e rotinas do utilizador, entre outros.
- **Compatibilidade com Dispositivos**: Verificar compatibilidade com outros tipos de dispositivos, além dos que foram utilizados para desenvolvimento do sistema.

5.3 Conclusão

Este trabalho teve como objetivo o desenvolvimento de uma aplicação de Automação Residencial. Foram estudados os aspectos teóricos e tecnológicos que envolvem o tema, especificados o escopo, requisitos e as tecnologias a serem utilizadas no projeto e, por fim, a aplicação foi implementada. O resultado é um aplicativo compatível com a plataforma Linux, capaz de rodar em hardware de baixo custo como o Raspberry Pi. A aplicação utiliza a tecnologia *ZWave* para controle de dispositivos, e disponibiliza uma interface *web* para que o usuário possa acessá-la por meio de um navegador de internet.

O trabalho foi importante para a compreensão de várias tecnologias, dentre elas de Automação Residencial, comunicação entre processos e frameworks de desenvolvimento *web*. Além disso, o resultado final é um software funcional, de código aberto, que pode ser utilizado para automatização simples de uma residência.

Referências Bibliográficas

[1] HECKMAN, Davin. The Revolutions in Personal Computing and Origin of the Smart House. In: HECKMAN, Davin. **A Small World: Smart Houses and the Dream of the Perfect Day**. Los Angeles: Duke University Press, 2008. p. 55-56.

[2] BRUSH, A.j. Bernheim; LEE, Bongshin; MAHAJAN, Ratul. **Home Automation in the Wild: Challenges and Opportunities**. CHI 2011 Advance Technical Conference Program. Vancouver, p. 2115-2124. maio 2011.

[3] DENNIS, Andrew K. What home automation is. In: DENNIS, Andrew K. **Raspberry Pi Home Automation with Arduino**. Birmingham: Packt Publising, 2013. p. 17-19.

[4] FIBARO. **Advanced User's Guide**. Fibaro Home Intelligence, 2011.

[5] HCA Tech. **Home Control Assistant User Guide**, Version 12. HCA Tech, 2013.

[6] PULSE Work. **UPB Technology Descriptio**n, Version 1.4. Northridge: Pulse Work, 2005.

[7] Control4. **Control4 System User Guide**. Salt Lake City: Control4 Corporation, 2013.

[8] LinuxMCE Community. **The LinuxMCE wiki**. Disponível em: <http://wiki.linuxmce.org/index.php/Main_Page>. Acesso em: 25 jun. 2014.

[9] JEIHALA, Sergei. **MajorDoMo - Open-source home automation platform.** Disponível em: <http://majordomohome.com/>. Acesso em: 25 jun. 2014.

[10] GERSHENFELD, Neil; KRIKORIAN, Raffi; COHEN, Danny. **The Internet Of Things**. New York: Scientific American, 2004.

[11] WEI, Chao-huang; THAN, Hoang; WANG, Yu-ning. **Realization of Home Appliances Control System based on Power Line Communication Technology**. Taiwan: Southern Taiwan University, 2012.

[12] INSTEON. **Insteon Whitepaper: The Details**, Version 2.0. Irvine: INSTEON, 2013.

[13] KINNEY, Patrick. **ZigBee Technology: Wireless Control that Simply Works**. Communications Design Conference, 2013.

[14] IEEE Computer Society. **Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)**. New York: IEEE Computer Society, 2006.

[15] ZWave Alliance. **About ZWave Technology**. Disponível em: <http://www.z-wavealliance.org/about-z-wave>. Acesso em: 30 jun. 2014.

[16] AIRBERRY. **Whitepaper: Introduction to Mesh Networks**. Airberry, 2012.

[17] OpenZWave Community. **OpenZWave Library**. Disponível em: <http://www.openzwave.com/dev/>. Acesso em: 30 jun. 2014.

[18] HONEYWELL. **H316 General Porpouse Digital Computer**. Framingham: Honeywell, 1970.

[19] MOZER, Michael C. **Lessons from an Adaptive Home**. Colorado: University Of Colorado, 2005.

[20] DAVIDOFF, Scott; LEE, Min Kyung; YIU, Charles. Principles of Smart Home Control. **Ubicomp**. Carnegie Mellon, p. 16-34. 2006.

[21] ZHENG, Huiru; WANG, Haiying; BLACK, Norman. **Human Activity Detection in Smart Home Environment with Self-Adaptive Neural Networks**. Sanya: Ieee, 2008.

[22] NEßELRATH, Robert; LU, Chensheng; SCHULZ, Christian H. **A Gesture Based System for Context - Sensitive Interaction with Smart Homes**. Berlin: Springer, 2011.

[23] HAMILL, Melinda; YOUNG, Vicky; BOGER, Jennifer. **Development of an automated speech recognition interface for personal emergency response systems**. Toronto: Springer, 2009.

[24] TRUONG, Khai N.; HUANG, Elaine M.; ABOWD, Gregory D. **CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home**. Atlanta: Georgia Institute Of Technology, 2004.

[25] KÜHNE, Christine; WESTERMANN, Tilo; HEMMERT, Fabian. **I'm home: Defining and evaluating a gesture set for smart-home control**. Berlin: Elsevier, 2011.

[26] bibi21000. **Python OpenZWave Documentation**. Disponível em: <http://bibi21000.gallet.info/index.php/en/home-automation-uk/126-python-openzwave-documentation.html>. Acesso em: 15 Set. 2014.

[27] HINTJENS, Pieter. **ZeroMQ Code Connected**. iMatix Corporation and Contributors. Disponível em: <http://zguide.zeromq.org/>. Acesso em: 15 Set. 2014.

[28] ZeroMQ. **ZeroMQ Language Bindings**. iMatix Corporation and Contributors. Disponível em: <http://zeromq.org/bindings:_start>. Acesso em: 15 Set. 2014.

[29] Joyent, Inc. **About Node.js**. Disponível em: <http://nodejs.org/about/>. Acesso em: 15 Set. 2014.

[30] Google. **AngularJS - Superheroic JavaScript MVW Framework**. Disponível em: <https://angularjs.org/>. Acesso em: 15 Set. 2014.

[31] Apache Software Foundation. **Apache Thrift**. Disponível em: <http://thrift.apache.org/>. Acesso em: 16 Set. 2014.

[32] Google. **Protocol Buffers**. Disponível em: <https://developers.google.com/protocol-buffers/?csw=1>. Acesso em: 16 Set. 2014.

[33] SQLite. **SQLite Home Page**. Disponível em: <http://www.sqlite.org/>. Acesso em: 16 Set. 2014.

[34] Raspbian. **Raspbian**. Disponível em: <http://raspbian.org/>. Acesso em: 17 Set. 2014.

[35] OpenZWave Wiki Pages. **Controller Compatibility List**. Disponível em: <https://code.google.com/p/open-zwave/wiki/Controller_Compatibility_List>. Acesso em: 17 Set. 2014.

[36] SOMMERVILLE, Ian. Chapter 4: Requirements engineering. In: SOMMERVILLE, Ian. **Software Engeneering.** St Andrews: Pearson Education, 2011. sec. 4.1.

[37] HINTJENS, Pieter. **ZeroMQ - The Guide**. Disponível em: <http://zguide.zeromq.org/page:all>. Acesso em: 23 set. 2014.

[38] CHACON, Scott. **Git**. Disponível em: <http://git-scm.com/>. Acesso em: 24 set. 2014.

[39] WIRTH, Michael; MCCUAIG, Judi. **Making Programs With The Raspberry Pi**. Richmond: Acm, 2014.

Anexos

Anexo A

Scripts utilizados

A.1 Instalação de dependências

```
#############################
# Dependencies Installation #
###     for Raspbian     #####
#
#
# First, on /etc/sudoers, add:
# > Defaults env_keep += "PYTHONPATH"
mkdir -p ~/software
# Zero MQ
sudo apt-get install uuid-dev libtool autoconf automake -y
cd ~/software/
wget http://download.zeromq.org/zeromq-3.2.4.tar.gz
tar xvfz zeromq-3.2.4.tar.gz
cd zeromq-3.2.4
./configure
make
sudo make install
# Installing Python ZMQ binding
cd ~/software/
wget http://pypi.python.org/packages/source/C/Cython/Cython-0.16.tar.gz
tar xvfz Cython-0.16.tar.gz
cd Cython-0.16
sudo python setup.py install
cd ~/software/
git clone git://github.com/zeromq/pyzmq.git
```

```
cd pyzmq
./setup.py configure
sudo ./setup.py install
# or sudo pip install pyzmq
# SQLite3
sudo apt-get install sqlite3 -y
sudo apt-get install libsqlite3-dev -y
# Lua dependencies: luarocks, lua-sqlite3, luajson,
sudo apt-get install luarocks -y
sudo luarocks install luajson
sudo luarocks install lsqlite3
sudo luarocks install lzmq  # requires zmq 3.2.4
# Add new line: /usr/local/lib:
# > sudo vi /etc/ld.so.conf
# > sudo ldconfig
# Python Open ZWave + dependencies. Note: needs cython v0.14
cd ~/software
sudo apt-get install python-pip python-dev -y
sudo pip install cython==0.14
sudo apt-get install python-dev python-setuptools python-louie -y
sudo apt-get install build-essential libudev-dev g++ make -y
wget http://bibi21000.no-ip.biz/python-openzwave/python-openzwave-0.2.6.tgz
tar -zxvf python-openzwave-0.2.6.tgz
cd python-openzwave-0.2.6
mv openzwave packed_openzwave
# download and compile OpenZWave
wget http://www.openzwave.com/downloads/openzwave-1.0.791.tar.gz
tar -zxvf openzwave-1.0.791.tar.gz
mv openzwave-1.0.791 openzwave
cd openzwave
make
sudo make install
cd ..
# compile Python OpenZWave
./compile.sh
sudo ./install.sh
Node.js
# Node.js
wget http://node-arm.herokuapp.com/node_latest_armhf.deb
```

```
sudo dpkg -i node_latest_armhf.deb
cd ~/eaZy/apps/ui/eazy_web/
sudo npm install
```

A.2 Variáveis de ambiente

```
export EAZYPATH='/home/matheus/projects/eaZy/'
export PYTHONPATH=$PYTHONPATH:$EAZYPATH/lib/bus/
```

A.3 *Deploy* do sistema

```
#/bin/bash
echo ''
echo 'Deploying to pi@eazy:/home/pi/eaZy/'
echo ''
HOST="eazy.local" # Default for RPi is 'raspberrypi.local'
echo "checking host $HOST..."
ping -q -w 5 $HOST
if [ $? -ne 0 ]; then
    echo "error!! problem accessing host."
    exit 1;
fi
echo " done."
echo 'Backing up old files (if any)...'
ssh pi@$HOST "mkdir -p ~/tmp/ && rm -rf ~/tmp/eaZy_backup
              && mv ~/eaZy ~/tmp/eaZy_backup && mkdir -p ~/eaZy"
echo ' done.'
echo 'Installing new files...'
scp -r ../* pi@$HOST:~/eaZy
echo ' done.'
echo 'Installing dependencies... Oh boy, this can take a while. Maybe some coffee?'
ssh pi@$HOST "cd ~/eaZy/ && mv scripts/eaZy.sh .
              && cd ~/eaZy/apps/ui/eazy_web
              && rm -rf node_modules && npm install"
echo ' done.'
exit 0;
```

A.4 Inicialização do sistema

Script de Inicialização do eaZy

```
echo "Running eaZy!"
# Checking if the paths are set
if [ -z "$PYTHONPATH" ]; then
    echo "Please export environment variable PYTHONPATH. Example:"
    echo 'export PYTHONPATH=$PYTHONPATH:$EAZYPATH/lib/bus/'
    exit 1
fi
# Getting IP
myip=`ifconfig | grep -Eo 'inet (addr:)?([0-9]*\.){3}[0-9]*' |
        grep -Eo '([0-9]*\.){3}[0-9]*' | grep -v '127.0.0.1'`
# Initiating eaZy components
node apps/ui/eazy_web/app.js &> /dev/null &
lua apps/home_stack/home_stack.lua &> /dev/null &
sudo python apps/device_controller/device_controller.py &> /dev/null &
echo ""
echo "All apps are running. Give about 5 minutes for all systems"
echo "to be available, as ZWave takes some time to be ready."
echo ""
echo "Address: http://eazy.local:3000 or http://$myip:3000"
```

Trigger para Iniciar o eaZy com a Plataforma

```
pi@eazy ~ $ cat /etc/rc.local
#!/bin/sh -e
# File: /etc/rc.local

# Print the IP address
_IP=$(hostname -I) || true
if [ "$_IP" ]; then
      printf "My IP address is %s\n" "$_IP"
fi

echo "But you can access me with eazy.local :]"
echo ""

echo "Initializing EAzY"
```

```
export EAZYPATH=/home/pi/eaZy
export PYTHONPATH=$PYTHONPATH:$EAZYPATH/lib/bus/

cd /home/pi/eaZy/
./eaZy.sh
```

Anexo B

Código Fonte

B.1 Biblioteca de Comunicação por Barramento

client.lua

```lua
--------------------------------------------------------------------------------
-- Class used by the Client entity to communicate to the Server.
--
-- The communication channel should be configured using the three ports:
-- - com_port: Used to receive broadcast messages from the Server entity.
-- - set_port: Used to send messages/request data to the Server entity.
-- - res_port: Used to receive a responce from a Server.
--------------------------------------------------------------------------------
local bus_client = {}
local zmq  = require('lzmq')
local json = require('json')
local util = require('lib.util.util')
--------------------------------------------------------------------------------
-- Default Configuration
--------------------------------------------------------------------------------
local HOST     = '127.0.0.1'
local COM_PORT = 5556
local SET_PORT = 5557
local RES_PORT = 5558
--------------------------------------------------------------------------------


--- Creates a new Bus Client instance.
--
-- @param opt An object that contains the configuration for this Bus Client. If
```

```
--          not provided, the default configurations will be set.
--          eg:
--          {id="Me", filter="server" host="localhost",
--          com_port=1, set_port=2, res_port=3}
-- @return table The new Bus Client instance.
function bus_client:new(opt)
    local self = {}
    setmetatable(self, {__index=bus_client})
    opt = opt or {}
    self.id       = opt.id       or 'client_id'
    self.host     = opt.host     or HOST
    self.com_port = opt.com_port or COM_PORT
    self.set_port = opt.set_port or SET_PORT
    self.res_port = opt.res_port or RES_PORT
    self.filter   = opt.filter   or nil
    return self
end

--- Prepares this Bus Server to be used.
-- Before sending/receiving messages, the method setup() should be called to
-- properly setup the socket configurations.
--
-- @return table The instance itself.
function bus_client:setup()
    self.context = zmq.context()
    self.sub_socket, err = self.context:socket(zmq.SUB)
    zmq.assert(self.sub_socket, err)
    if self.filter then self.sub_socket:subscribe(self.filter) end
    self.sub_socket:connect('tcp://'..self.host..':'..self.com_port)
    return self
end

--- Receives a message from the communication channel's Server
-- Tryies to get a message from the communication channel, checking the
-- 'com_port' for broadcast messages from the Server.
--
-- @param blocking If false, the method will check if there is a message and
--          then retrun the message, if it exists, or 'nil' if no message was
--          received.  If true, the method will block the interpreter until a new
```

```
--        message arrives, which then is returned.
-- @return string The message, if exists, or nil, if no message was received.
function bus_client:check_income(blocking)
    local raw_data = self.sub_socket:recv(blocking and nil or zmq.NOBLOCK)
    if not raw_data then return nil end
    local sender, msg = unpack(util.split(raw_data, ' ',true))
    return json.decode(msg), sender
end

--- Send a message to the Server.
-- Send the given message to the Server of this communication channel, using the
-- 'set_port'.
--
-- @param msg A table or string containing the message.
-- @return table The instance itself.
function bus_client:send(data, type)
    if not data then return end
    local msg = {type=type or 'send', data=data, sender=self.id}
    local set_socket, err = self.context:socket(zmq.PAIR)
    zmq.assert(set_socket, err)
    set_socket:connect('tcp://'..self.host..':'..self.set_port)
    set_socket:send(json.encode(msg))
    set_socket:close()
    return self
end

--- Make a request for the Server.
-- When called, a message is sent to the Server indicating this Client has made
-- a request. The Client will stay blocked until the response from the Server is
-- received on the 'res_port', and then returned.
--
-- @param request A string indicating the request (eg. 'device_list')
-- @return table The response from the Server.
function bus_client:get(request)
    if not request then return end
    self:send(request, 'get')
    local res_socket, err = self.context:socket(zmq.PAIR)
    zmq.assert(res_socket, err)
    res_socket:bind('tcp://'..self.host..':'..self.res_port)
```

```
    local response = res_socket:recv()
    res_socket:close()
    return json.decode(response).data
end

return bus_client
```

server.lua

```
--------------------------------------------------------------------------------
-- Class used by the Server entity to control communication between components.
--
-- The communication channel should be configured using the three ports:
-- - com_port: Used to broadcast message to Client entities.
-- - set_port: Used by Clients to send messages to the Server entity.
-- - res_port: Used by the Servery entity to respond to a Client request.
--------------------------------------------------------------------------------
local bus_server = {}
local zmq  = require('lzmq')
local json = require('json')
--------------------------------------------------------------------------------
-- Default Configuration
--------------------------------------------------------------------------------
local HOST     = '127.0.0.1'
local COM_PORT = 5556
local SET_PORT = 5557 -- clients send to this port
local RES_PORT = 5558 -- clients get on this port
--------------------------------------------------------------------------------


--- Creates a new Bus Server instance.
--
-- @param opt An object that contains the configuration for this Bus Server. If
--        not provided, the default configurations will be set.  eg: {id="My
--        Server", host="localhost", com_port=1, set_port=2, res_port=3}
-- @return table The new Bus Server instance.
function bus_server:new(opt)
    local self = {}
    setmetatable(self, {__index=bus_server})
    opt = opt or {}
    self.id       = opt.id       or 'server_id'
```

```
    self.host     = opt.host     or HOST
    self.com_port = opt.com_port or COM_PORT
    self.set_port = opt.set_port or SET_PORT
    self.res_port = opt.res_port or RES_PORT
    return self
end

--- Prepares this Bus Server to be used.
-- Before sending/receiving messages, the method setup() should be called to
-- properly setup the socket configurations.
--
-- @return table The instance itself.
function bus_server:setup()
    self.context = zmq.context()

    self.pub_socket, err = self.context:socket(zmq.PUB)
    zmq.assert(self.pub_socket, err)
    self.pub_socket:bind('tcp://'..self.host..':'..self.com_port)

    self.set_socket, err = self.context:socket(zmq.PAIR)
    zmq.assert(self.set_socket, err)
    self.set_socket:bind('tcp://'..self.host..':'..self.set_port)

    return self
end

--- Distribute a message to all Clients.
-- When called, the message will be distributed to all Clientes connected in
-- this communication channel, using the 'com_port'.
--
-- @param msg A table containing the message.
--        eg: {sender="Me", type="hello", data="from server"}
-- @return table The instance itself.
function bus_server:distribute(msg)
    if not msg then return end
    self.pub_socket:send(msg.sender .. ' ' .. json.encode(msg))
    print('distribute:',msg.sender .. ' ' .. json.encode(msg))
    return self
end
```

```
--- Receives a message from the communication channel.
-- Tryies to get a message from the communication channel, checking the
-- 'set_port' for new messages.
--
-- @param noblocking If true, the method will check if there is a message and
--        then return the message, if it exists, or 'nil' if no message was
--        received.  If false, the method will block the interpreter until a new
--        message arrives, which then is returned.
-- @return string The message, if exists, or nil, if no message was received.
function bus_server:getMessage(noblocking)
    local msg = self.set_socket:recv(noblocking and zmq.NOBLOCK)
    if not msg then return nil end
    return json.decode(msg)
end

--- Respond to a Client request.
-- When a Client has requested (ie. sent a message with type='get'), it will be
-- wainting for a response through the 'res_port'. This method uses the
-- 'res_port' to respond to a client request directly.
--
-- @param msg A table containing the message.
--        eg: {sender="Me", type="get", data="user_list"}
-- @return table The instance itself.
function bus_server:sendResponse(msg)
    local res_socket, err = self.context:socket(zmq.PAIR)
    zmq.assert(res_socket, err)
    res_socket:connect('tcp://'..self.host..':'..self.res_port)
    res_socket:send(json.encode(msg))
    res_socket:close()
    return self
end

return bus_server
```

B.2 Componente Home Stack

home_stack.lua

```lua
--------------------------------------------------------------------------------
-- Home Stack
--
-- Module responsible for interfacing User Interface and Device Controller.
--
-- The objective of Home Stack is to provide a seamless interface through where
-- any UI implementation can access (using Zero MQ) to get user information.
-- Any user preferences, device names and scenes are stored in here.
-- Also, all events trigged by scenes are sent to UI from Home Stack.
--------------------------------------------------------------------------------

local json          = require('json')
local bus_server    = require('lib.bus.server')

local scene_engine  = require('apps.home_stack.scene_engine')
local device_mapper = require('apps.home_stack.device_mapper')
local db            = require('apps.home_stack.data.db')

--------------------------------------------------------------------------------
-- Local functions
--------------------------------------------------------------------------------

local lifeLoop
local onUIMessage
local onDeviceMessage
local getDeviceResponse
local log = function(...) print('<:> Home Stack <:>', ...) end

--------------------------------------------------------------------------------
-- Setup
--------------------------------------------------------------------------------

local device = bus_server:new({
    id     = 'home_stack',
    filter = 'device_controller'
})
```

```
local ui = bus_server:new({
    id       = 'home_stack',
    com_port = 5560,
    set_port = 5561,
    res_port = 5562
})

local running = true

device:setup()
ui:setup()

--------------------------------------------------------------------------------
-- Home Stack
--------------------------------------------------------------------------------

function lifeLoop()
    local msg
    msg = device:getMessage('noblock')
    if msg then onDeviceMessage(msg); msg = nil end
    msg = ui:getMessage('noblock')
    if msg then onUIMessage(msg) end
end

function onDeviceMessage(msg)
    log('Message: sender,type,data', msg.sender, msg.type, msg.data)
    if msg.type == 'get' then
        device:sendResponse({
            data='foo'
        })
    elseif msg.type == 'send' then
        local evt = msg.data;

        if evt.type == 'update' then
            evt.data = device_mapper.mapToUI(evt.data)
            evt.id = evt.data.id
        end
        ui:distribute({
            sender = 'home_stack',
```

```
            type    = msg.type,
            data    = evt
        })
        if evt.type == 'update' then
            local event = scene_engine.check(evt.data)
            if event then
                event.id = device_mapper.getRawDeviceId(event)
                device:distribute({
                    sender = 'home_stack',
                    type    = msg.type,
                    data    = event
                })
            end
        end
    else
        ui:distribute({
            sender = 'home_stack',
            type    = msg.type,
            data    = msg.data
        })
    end
end

function onUIMessage(msg)
    log('UI Message: ', msg.sender, msg.type, json.encode(msg.data))
    if msg.type == 'get' then
        if msg.data == 'devicelist' then
            local list = getDeviceResponse('devicelist')
            ui:sendResponse(device_mapper.mapToUI(list.data))
        elseif msg.data == 'scenelist' then
            local list = scene_engine.getScenes()
            ui:sendResponse(list)
        end
    elseif msg.type == 'send' then
        local evt = msg.data;
        if evt.type == 'updatedevice' then
            device_mapper.updateDevice(evt.data)
        elseif evt.type == 'deletedevice' then
            device_mapper.deleteDevice(evt.data)
```

```
        elseif evt.type == 'addscene' then
            scene_engine.add(evt.data)
        elseif evt.type == 'deletescene' then
            scene_engine.delete(evt.data)
        elseif evt.type == 'setstate' then
            msg.data.id = device_mapper.getRawDeviceId(msg.data)
            device:distribute({
                sender = 'home_stack',
                type   = msg.type,
                data   = msg.data
            })
        end
    end
end

function getDeviceResponse(get)
    device:distribute({
        sender = 'home_stack',
        type   = 'get',
        data   = get
    })
    local got = false;
    local msg
    while not got do
        msg = device:getMessage()
        if msg.type == 'response' then got = true
        else onDeviceMessage(msg) end
    end
    return msg
end

log('Running ...')
while running do lifeLoop() end
log('Stopped.')
```

device_mapper.lua

```
--------------------------------------------------------------------------------
-- Device Mapper
--
```

```lua
-- This module manages UI devices by mapping them to real devices retrieved
-- from Device Controller.
-- The mapping is important so that the 'raw' devices retrieved by the
-- device controller can be turned into 'ui' devices, containing user
-- preferences for that specific device such as a custom device name.
--
-- If a device is unknown to the Device Mapper, it will be returned as the 'raw'
-- device plus an attribute 'registration', set to 'unregistered'.
--------------------------------------------------------------------------------

local device_mapper = {}
local db = require('apps.home_stack.data.db')

--------------------------------------------------------------------------------
-- Maps a Raw Device List (eg. list got from Device Controller with real device
-- ID's) to a UI Device List (eg. devices with ui custom attributes).
--
-- @param raw_list A device or device list retrieved directly from
-- device_controller.
--------------------------------------------------------------------------------

function device_mapper.mapToUI(raw_list)
    if not raw_list then return {} end
    local db_devices = db.getDevices()

    local fixDevice = function(device)
        -- get the ui device
        local ui_prefs = false
        for k, d in pairs(db_devices) do
            if device.id == d.id_device then ui_prefs = d end
        end
        if not ui_prefs then
            -- unknown device
            device.registration = 'unregistered'
            -- fix id
            device.id_device = device.id
            device.id = nil
        else
            -- fill device with ui settings
```

```
        for key, value in pairs(ui_prefs) do
            if key ~= 'type' then
                device[key] = value
            end
        end
    end
    -- fix strings to numeric values
    if device.consumption_accumulated then
        device.consumption_accumulated =
                tonumber(device.consumption_accumulated)
    end
    if device.consumption_current then
        device.consumption_current =
                tonumber(device.consumption_current)
    end
    if device.temperature then
        device.temperature = tonumber(device.temperature)
    end
    if device.luminance then
        device.luminance = tonumber(device.luminance)
    end
    return device
end

if #raw_list == 0 and next(raw_list) then -- single device
    return fixDevice(raw_list)
else -- device list
    local list = {}
    for k, device in pairs(raw_list) do
        if device.id ~= 25488113006 then
            table.insert(list, fixDevice(device))
        end
    end
    -- check if any device known by the DB is missing
    for k, db_dev in ipairs(db_devices) do
        local found = false
        for j, dev in ipairs(list) do
            if db_dev.id_device == dev.id_device then
                found = true
```

```
                end
            end
            if not found then -- if not, device to list
                db_dev.registration = 'missing'
                table.insert(list, db_dev)
            end
        end
        return list
    end
end

--------------------------------------------------------------------------------
-- Updates the device in Home Stack, setting new user preferences to it. If the
-- device is present on the database (identified by 'id_device') it is updated,
-- and if not, it is added.
--
-- @param device The device object with updated values (should contain the
-- device UI id.
--------------------------------------------------------------------------------

function device_mapper.updateDevice(device)
    if not device or not device.id_device or not device.type
            or not device.name then
        return print('ERR: Cannot update device.')
    end
    local stored = db.getDevice(device.id_device)
    if stored and next(stored) then
        db.updateDevice(device)
    else
        db.addDevice(device)
    end
end

--------------------------------------------------------------------------------
-- Deletes a device, removing it from the HomeStack database.
--
-- @param device The device object to be deleted from the database.
--------------------------------------------------------------------------------
```

```
function device_mapper.deleteDevice(device)
    if not device or not device.id or not device.id_device then
        return nil
    end
    db.deleteDevice(device)
end

-------------------------------------------------------------------------------
-- Gets a Raw Device ID.
--
-- @param device The UI Device.
-- @return number The raw Device ID, used by Device Controller.
-------------------------------------------------------------------------------

function device_mapper.getRawDeviceId(device)
    if not device then return end
    if device.id_device then return device.id_device end
    local id = device.id
    for _, db_dev in pairs(db.getDevices()) do
        if id == db_dev.id then
            return db_dev.id_device
        end
    end
end

return device_mapper
```

scene_engine.lua

```
-------------------------------------------------------------------------------
-- Scene Engine
--
-- Module responsible for acknowledging and creating actions for user defined
-- scenes.
-------------------------------------------------------------------------------

local scene_engine = {}
local db = require('apps.home_stack.data.db')

local cached_scenes
```

```lua
local db_updated = false

--------------------------------------------------------------------------------
-- Returns all scenes, as an array of Scene tables.
--
-- @return table The scene list.
--------------------------------------------------------------------------------

function scene_engine.getScenes()
    if db_updated or not cached_scenes then
        cached_scenes = db.getScenes()
    end
    db_updated = false
    return cached_scenes
end

--------------------------------------------------------------------------------
-- Adds a scene to the database.
--
-- @param scene The new scene to be added. Must be a Scene table.
--------------------------------------------------------------------------------

function scene_engine.add(scene)
    if not scene then return end
    if scene.source_device and scene.source_attr and scene.source_value
            and scene.target_device and scene.target_state then
        db.addScene(scene)
        db_updated = true
    end
end

--------------------------------------------------------------------------------
-- Deletes a scene from the database.
--
-- @param scene The scene to be deleted
--------------------------------------------------------------------------------

function scene_engine.delete(scene)
    if scene and scene.id then
```

```
        db.deleteScene(scene)
        db_updated = true
    end
end

--------------------------------------------------------------------------------
-- Checks the device states to verify if there is any associated scenes.
--
-- @param device The device with updated attributes to be checked.
-- @return table The scene event, if any, with the type 'setstate' to be sent to
-- the device controller.
--------------------------------------------------------------------------------

function scene_engine.check(device)
    local scenes = scene_engine.getScenes()
    local scene = nil
    local event = nil
    for _, s in pairs(scenes) do
        if s.source_device == device.id then
            scene = s
        end
    end
    if not scene then return nil end
    if not device[scene.source_attr] then return nil end
    if device[scene.source_attr] == scene.source_value then
        event = {
            id    = scene.target_device,
            state = scene.target_state,
            type  = "setstate"
        }
    end
    return event
end

return scene_engine
```

data/db.lua

```
--------------------------------------------------------------------------------
-- db
```

```lua
--
-- This module provides database access to Home Stack.
--
-- The Home Stack database is used to store only device customizations made by
-- the user, such as a custom device name. The database does not contain
-- information about the device itself, such as device capabilities.
--------------------------------------------------------------------------------

local db = {}
local sqlite3 = require('lsqlite3')

--------------------------------------------------------------------------------
-- Creates database (if needed) and loads device data to memory.
--------------------------------------------------------------------------------

function db.setup()
    local con   = assert(sqlite3.open('apps/home_stack/data/data.db'),
            "Could not open 'data.db'")
    local file  = assert(io.open('apps/home_stack/data/setup.sql'),
            "Database setup.sql not found!")
    local query = ""
    for q in file:lines() do query = query .. q end
    con:exec(query)
    file:close()
    db.con = con
end

--------------------------------------------------------------------------------
-- Returns a list of the devices on the database.
-- Each item on the list is in the form {id=#,id_device=#,name=""}
--
-- @return table The device list.
--------------------------------------------------------------------------------

function db.getDevices()
    local select_stmt = assert(db.con:prepare("SELECT * FROM device;"))
    local devices = {}
    for row in select_stmt:nrows() do
        table.insert(devices, row)
```

```
    end
    return devices
end

--------------------------------------------------------------------------------
-- Returns the device with the given id_device, if exists.
--
-- @param id_device The real device id of the device object.
-- @return table The device retrieced from database, if exists.
--------------------------------------------------------------------------------

function db.getDevice(id_device)
    if not id_device then return nil end
    local query = 'SELECT * FROM device WHERE id_device = '..id_device..';'
    local select_stmt = assert(db.con:prepare(query))
    local device = {}
    for row in select_stmt:nrows() do
        table.insert(device, row)
    end
    return device
end

--------------------------------------------------------------------------------
-- Adds a device to the database. The device must be a table, containing at
-- least {id_device=#,name="",type=""}.
--
-- @param device The device to be added.
-- @return boolean True if command was successfully executed, false otherwise.
--------------------------------------------------------------------------------

function db.addDevice(device)
    local query = 'INSERT INTO device ("id_device", "name", "type") '
                ..'VALUES ('..device.id_device..',"'..device.name..'","'
                ..device.type..'");'
    db.con:exec(query)
    return true
end

--------------------------------------------------------------------------------
```

```
-- Updates a device in thedatabase. The device must be a table, containing at
-- least {id_device=#,name="",type=""}.
--
-- @param device The device to be updated.
-- @return boolean True if command was successfully executed, false otherwise.
--------------------------------------------------------------------------------

function db.updateDevice(device)
    local query = 'UPDATE device '
                ..'SET name = "'..device.name..'", type = "'..device.type..'" '
                ..'WHERE id_device = '..device.id_device..';'
    db.con:exec(query)
    return true
end

--------------------------------------------------------------------------------
-- Deletes a device in the database. The device must be a table, containing at
-- least {id_device=#}.
--
-- @param device The device to be deleted.
-- @return boolean True if command was successfully executed, false otherwise.
--------------------------------------------------------------------------------

function db.deleteDevice(device)
    local query = 'DELETE FROM device WHERE id_device = '..device.id_device..';'
    db.con:exec(query)
    return true
end

--------------------------------------------------------------------------------
-- Returns a list of the scene on the database.
-- Each item on the list is a scene table.
--
-- @return table The list of scenes.
--------------------------------------------------------------------------------

function db.getScenes()
    local select_stmt = assert(db.con:prepare("SELECT * FROM scene;"))
    local scenes = {}
```

```
    for row in select_stmt:nrows() do
        table.insert(scenes, row)
    end
    return scenes
end

--------------------------------------------------------------------------------
-- Adds a scene to the database. The device must be a table, like:
--   {source_device=#,source_attr="",source_value="",target_device=#,
--     target_state=""}.
--
-- @param scene The scene to be added.
-- @return boolean True if command was successfully executed, false otherwise.
--------------------------------------------------------------------------------

function db.addScene(scene)
    local query = 'INSERT INTO scene ("source_device", "source_attr", '
                ..'"source_value", "target_device", "target_state") VALUES ('
                ..scene.source_device.. ',"'
                ..scene.source_attr..   '","'
                ..scene.source_value..  '",'
                ..scene.target_device.. ',"'
                ..scene.target_state..  '");'
    db.con:exec(query)
    return true
end

--------------------------------------------------------------------------------
-- Deletes a scene in the database. The param must be the scene table, which
-- contains the scene database id.
--
-- @param scene The scene to be deleted.
-- @return boolean True if command was successfully executed, false otherwise.
--------------------------------------------------------------------------------

function db.deleteScene(scene)
    if not scene or not scene.id then return false end
    local query = 'DELETE FROM scene WHERE id = '..scene.id..';'
    db.con:exec(query)
```

```
    return true
end

db.setup()
return db
```

data/setup.sql

```
CREATE TABLE IF NOT EXISTS device (
    id INTEGER PRIMARY KEY,
    id_device INTEGER NOT NULL,
    name VARCHAR(80) NOT NULL,
    type VARCHAR(80) NOT NULL
);

CREATE TABLE IF NOT EXISTS scene (
    id INTEGER PRIMARY KEY,
    source_device INTEGER NOT NULL,
    source_attr VARCHAR(80) NOT NULL,
    source_value VARCHAR(80) NOT NULL,
    target_device INTEGER NOT NULL,
    target_state VARCHAR(80) NOT NULL
);
```

B.3 Componente Device Controller

device_controller.py

```
import logging
try:
    from client import BusClient
except ImportError:
    print('WARNING: Please set PYTHONPATH correcly. See readme for more info.')

from zwave_controller import ZWaveController

running = True
logging.basicConfig(level=logging.DEBUG)

def lifeLoop():
```

```
    msg = home_stack.check_income()
    if msg:
        onHomeStackMessage(msg[0]);

def onHomeStackMessage(msg):
    logging.info('Message: from,type,data', extra=msg)
    if msg['type'] == 'get':
        if msg['data'] == 'devicelist':
            devicelist = zwave.getDeviceList()
            home_stack.send(devicelist, 'response')
    elif msg['type'] == 'send':
        device_id = msg['data']['id']
        mtype = msg['data']['type']
        state = msg['data']['state']
        if mtype == 'setstate':
            zwave.setDeviceState(device_id, state)
            print('New state of ' + str(device_id) + ' is ' + state)

def onDeviceUpdate(device):
    print('will send notifycation!')
    msg = {'type':'update','data':device,'id':device['id']}
    home_stack.send(msg, 'send')

home_stack = BusClient('device_controller', 'home_stack')
home_stack.setup()

zwave = ZWaveController()
zwave.setup(onDeviceUpdate)

logging.info('Running ...')
while running:
    lifeLoop()
logging.info('Stopped.')
```

zwave_controller.py

```
import sys, os

import openzwave
from openzwave.node import ZWaveNode
```

```
from openzwave.value import ZWaveValue
from openzwave.scene import ZWaveScene
from openzwave.controller import ZWaveController
from openzwave.network import ZWaveNetwork
from openzwave.option import ZWaveOption
from louie import dispatcher, All
from threading import Timer


class ZWaveController():
    network = None

    def setup(self, updateCallback):
        dispatcher.connect(self.onNetworkReady, ZWaveNetwork.SIGNAL_NETWORK_READY)
        dispatcher.connect(self.onNetworkStart, ZWaveNetwork.SIGNAL_NETWORK_STARTED)
        dispatcher.connect(self.onNetworkFailed, ZWaveNetwork.SIGNAL_NETWORK_FAILED)

        # TODO: make udev.symlink rule to a specific port (USB0/1)
        # Uncomment this to run on PC (remember to update the zwave config path)
        #options = ZWaveOption("/dev/ttyUSB0", \
        #  config_path="/home/<USER>/software/python-openzwave-0.2.6/openzwave/config", \
        options = ZWaveOption("/dev/ttyUSB0", \
          config_path="/home/matheus/software/python-openzwave-0.2.6/openzwave/config", \
          user_path=".", cmd_line="")
        options.set_append_log_file(False)
        options.set_console_output(False)
        options.set_save_log_level('Debug')
        options.set_poll_interval(30);
        options.set_suppress_value_refresh(False)
        options.addOptionBool("AssumeAwake", True)
        options.set_logging(False)
        options.lock()
        self.network = ZWaveNetwork(options, autostart=False)
        self.onDeviceUpdateCallback = updateCallback
        self.network.start()
        self.addedConnections = False
        Timer(2*60, self.setupConnections).start()
```

```
def tearDown(self):
    network.stop()

def getDeviceList(self):
    devices = []
    for node in self.network.nodes:
        if node == 1: continue # don't add the controller
        devices.append(self.buildDevice(node))
    return devices

def buildDevice(self, node):
    dev = {}
    dev['id'] = int(self.network.home_id)*1000 + node
    dev['type'] = 'unknown'
    dev['product_name'] = self.network.nodes[node].product_name
    if self.getValueForLabel(node, 'Switch'):
        dev['type'] = 'appliance'
        val = self.getValueForLabel(node, 'Energy')
        dev['consumption_accumulated'] = type(val) != "None" and val or 0
        val = self.getValueForLabel(node, 'Power')
        dev['consumption_current'] =  type(val) != "None" and val or 0
        if self.getValueForLabel(node, 'Switch') == 'True':
            dev['state'] = 'on'
        else:
            dev['state'] = 'off'
    if self.getValueForLabel(node, 'Sensor'):
        dev['type'] = 'sensor'
        dev['temperature'] = self.getValueForLabel(node, 'Temperature')
        dev['luminance']   = self.getValueForLabel(node, 'Luminance')
        dev['presence']    = "undetected"
    dev['battery_level'] = self.getValueForLabel(node, 'Battery Level')
    return dev

def getValueForLabel(self, node, label):
    for v in self.network.nodes[node].values:
        if self.network.nodes[node].values[v].label == label:
            #self.network.nodes[node].refresh_value(v);
            return str(self.network.nodes[node].values[v].data_as_string)
    return None
```

```
def setDeviceState(self, device_id, state):
    node = device_id%1000
    if not self.network.nodes[node]: return
    for val in self.network.nodes[node].get_switches() :
        self.network.nodes[node].set_switch(val, True if state=='on' else False)

def setupConnections(self):
    self.addedConnections = True
    dispatcher.connect(self.onNodeUpdate, ZWaveNetwork.SIGNAL_NODE)
    dispatcher.connect(self.onNodeUpdateValue, ZWaveNetwork.SIGNAL_VALUE)
    dispatcher.connect(self.onNodeUpdateValue, ZWaveNetwork.SIGNAL_NODE_EVENT)
    dispatcher.connect(self.onNodeUpdateValue, ZWaveNetwork.SIGNAL_VALUE_CHANGED)
    dispatcher.connect(self.onNodeUpdateValue, ZWaveNetwork.SIGNAL_VALUE_REFRESHED)

# Event Handlers

def onNetworkStart(self, network):
    print("network started : homeid %0.8x - %d nodes were found." % \
        (network.home_id, network.nodes_count))

def onNetworkFailed(self, network):
    print("network can't load :(")

def onNetworkReady(self, network):
    print("network : I'm ready : %d nodes were found." % network.nodes_count)
    print("network : my controller is : %s" % network.controller)
    self.network = network
    if not self.addedConnections:
        self.setupConnections()

def onNodeUpdate(self, network, node):
    print('node UPDAAAATEEE : %s.' % node)
    self.network = network

def onNodeUpdateValue(self, network, node, value):
    print('node : %s.' % node)
    print('value: %s.' % value)
    if node.node_id == 1: return # don't send controller notifications
```

```
dev = self.buildDevice(node.node_id)

if type(value) is int:
    if dev['type'] == 'sensor' and value == 255:
        dev['presence'] = 'detected'
        self.network = network
        self.onDeviceUpdateCallback(dev)

if type(value) is ZWaveValue:
    if dev['type'] == 'appliance' and value.label == 'Switch':
        state = value.data and 'on'  or 'off'
        dev['state'] = state
        self.network = network
        self.onDeviceUpdateCallback(dev)

    if dev['type'] == 'appliance' and value.label == 'Power':
        power = str(value.data)
        if dev['state'] == 'off' or (dev['state'] == 'on' and float(power) != 0):
            dev['consumption_current'] = power
            self.network = network
            self.onDeviceUpdateCallback(dev)
        else:
            self.network = network
            print('WHAATF do i do with this? %s', power)

    if dev['type'] == 'appliance' and value.label == 'Energy':
        energy = str(value.data)
        dev['consumption_accumulated'] = energy
        self.network = network
        self.onDeviceUpdateCallback(dev)

    if dev['type'] == 'sensor' and value.label == 'Temperature':
        temperature = str(value.data)
        dev['temperature'] = temperature
        self.network = network
        self.onDeviceUpdateCallback(dev)

    if dev['type'] == 'sensor' and value.label == 'Luminance':
        luminance = str(value.data)
```

```
            dev['luminance'] = luminance
            self.network = network
            self.onDeviceUpdateCallback(dev)

        if value.label == 'Battery Level':
            battery = str(value.data)
            dev['battery_level'] = battery
            self.network = network
            self.onDeviceUpdateCallback(dev)
```

B.4 Componente eaZy Web

app.js - Servidor

```
var express      = require('express');
var engine       = require('ejs-locals');
var path         = require('path');
var favicon      = require('static-favicon');
var logger       = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser   = require('body-parser');
var _            = require('lodash');

var app = express();
var server = require('http').Server(app);

// Bus and Socket.IO setup

var io = require('socket.io')(server);
zbus = require('./services/BusService.js');
zbus.setup(io);

// View engine setup

app.set('views', path.join(__dirname, 'views'));
app.engine('ejs', engine);
app.set('view engine', 'ejs');
app.use(favicon());
app.use(bodyParser.json());
```

```
app.use(bodyParser.urlencoded());
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

// Routes setup

var dashboard = require('./routes/dashboard');
var device    = require('./routes/device');
var energy    = require('./routes/energy');
var scene     = require('./routes/scene');
var settings  = require('./routes/settings');

app.get('*', dashboard);
app.use('/', dashboard);
app.use('/device',   device);
app.use('/energy',   energy);
app.use('/scene',    scene);
app.use('/settings', settings);


/// Catch 404 and forwarding to error handler
app.use(function(req, res, next) {
    var err = new Error('Not Found');
    err.status = 404;
    next(err);
});

// Development error handler
// Will print stacktrace
if (app.get('env') === 'development') {
    app.use(function(err, req, res, next) {
        res.status(err.status || 500);
        res.render('error', {
            message: err.message,
            error: err
        });
    });
}
```

```
// Production error handler
// No stacktraces leaked to user
app.use(function(err, req, res, next) {
    res.status(err.status || 500);
    res.render('error', {
        message: err.message,
        error: {}
    });
});

// Server setup
server.listen(process.env.PORT || 3000);
console.log('Listenning to port',process.env.PORT || 3000);
```

services/BusService.js

```
var zmq = require('zmq');

/*
 * Zero MQ Setup
 * Default CommStack settings
 */
var com_socket;

var HOST     = '127.0.0.1';
var COM_PORT = 5560;
var SET_PORT = 5561;
var RES_PORT = 5562;


module.exports = {

  setup: function(io) {
    com_socket = zmq.socket('sub');
    com_socket.connect('tcp://'+HOST+':'+COM_PORT);
    com_socket.subscribe('home_stack');
    // On Message callback
    com_socket.on('message', function(msg) {
      var msg = msg.toString();
      try {
```

```
        var sender = msg.split(' ',1)[0];
        var evt  = JSON.parse(msg.substr(msg.indexOf(' ')+1));
        console.log(evt);

        if(typeof evt.data == 'object') {
          io.sockets.emit(evt.data.type, evt.data);
        } else {
          console.log('dunno what to do.');
        }
    } catch(e) {
        console.log("Wrong message format. Msg:",msg,"Error:",e);
    }
  });
},

/*
 * Send a message to the Server. Send the given message to the Server of
 * this communication channel.
 */

sendMessage: function(data, type) {
  var msg = {type:type||'send', data:data||'', sender:'web'};
  var set_socket = zmq.socket('pair');
  set_socket.connect('tcp://'+HOST+':'+SET_PORT);
  console.log('sending', JSON.stringify(msg));
  set_socket.send(JSON.stringify(msg));
  set_socket.close();
},

/*
 * Make a request for the Server.
 * When called, a message is sent to the Server indicating this Client has made
 * a request.
 */

getData: function(data, cb) {
  this.sendMessage(data, 'get');
  var res_socket =  zmq.socket('pair');
  res_socket.bind('tcp://'+HOST+':'+RES_PORT);
```

```
    res_socket.on('message', function(response) {
      cb(null, JSON.parse(response.toString()));
      res_socket.close();
    });
  }
};
```

Anexo C

Artigo

Home Automation System based on Raspberry Pi and Open ZWave

Matheus Hoffmann Silva[1], **João Cândido Dovicchi**[1]

[1]Dpt. Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Postal 476 – 88040-900 – Florianópolis – SC – Brazil

hoffmann.matheus@gmail.com, dovicchi@inf.ufsc.br

Abstract. *Aiming at bringing benefits to the everyday living, it is usual to men to include technology into their lives. Concerning this, Home Automation is the technology that strives to improve the quality of peoples lives within their homes, be at helping on their daily affairs or making their environment more secure and comfortable as possible. The present paper exposes the processes of conception and development of a Home Automation system, which differs from others on its low-cost final solution, effortless configuration, and a flexible architecture that enables the use of multiple User Interfaces and Home Automation protocols.*

1. Introduction

The intense technological evolution on areas such as Human-Computer Interaction (HCI), Robotics and Artificial Intelligence (AI) enabled new ways and possibilities of communication between between machines and men. The so called Internet of Things (IoT) is but a reflex of these events and is a sample of how technology can modify and improve aspects of the everyday lives of people.

This paper will briefly explore the subjects concerning Home Automation and then propose a solution that focuses on the current downsides of this type of system: simple plug-and-play installation and configuration, and an affordable final cost. Note that this is a condensed work based on the thesis [5].

1.1. Home Automation

A Home Automation system is one that '*removes as many human interaction as possible, and desirable, from daily affairs*'[2]. That is, Home Automation systems aim at bringing the IoT benefits into the men day-to-day home lives, so they can be more comfortable and efficient.

The subject has been focus of studies for decades. A good example is the first Home Automation system documented, the ECHO IV [1]: in 1966 the north american engineer Jim Sutherland created the experimental ECHO IV, a bulky, multitasking home automation system. The system was capable of automatically compute the family finances and served as a message board, where family members could leave messages to one another.

In 1969, Honeywell released the H316 [3], known as the Kitchen Computer. The H316 was the first commercialized computer targeted to homes, and was able to store recipes and provide cooking tips. Though innovative, the Kitchen Computer was a failure due to its price of ten thousand dollars.

1.2. Device Communication Protocols

These kinds of systems, specially the commercially mature solutions, make use of intelligent devices that communicate through well consolidated protocols such as Insteon, ZigBee and ZWave. These protocols enable data exchange between sensors, appliances and other smart devices to communicate with each other, to create a network of home automation devices. A system that wishes to make use of such network, and therefore take control of it, needs to extend the communication protocol.

Each existent protocol has its pros and cons, so a study was made to choose the best technology to be used with EAZY. The technologies considered were: X10 [4], UPB [9], Insteon [6], ZigBee [7] and ZWave [8]. The Table 1 shows some characteristics of each technology.

Technology	Communication mean	Data Transmission Rate
X10	Power Lines	20 bit/s
UPB	Power Lines	240 bit/s
Insteon	Power Lines, Radio	(line): 12,8 kbit/s, (air): 37,5 kbit/s
ZigBee	Radio	250 kbit/s
ZWave	Radio	100 kbit/s

Table 1: Comparison of Home Automation Devices Communication Technologies.

The chosen technology was **ZWave**, due to its flexibility with a wireless *mesh* based network, meaningful number of devices available at the market and their average competitive price. Also, there is an open source alternative to build applications that make use of this protocol, which is *Open ZWave*.

Open ZWave [10] is an open source implementation of the ZWave protocol, built by reverse engineering and sniffing from device communication. The community backing the project is expressive and active, which can help with upcoming difficulties during its usage.

2. eaZy Concept

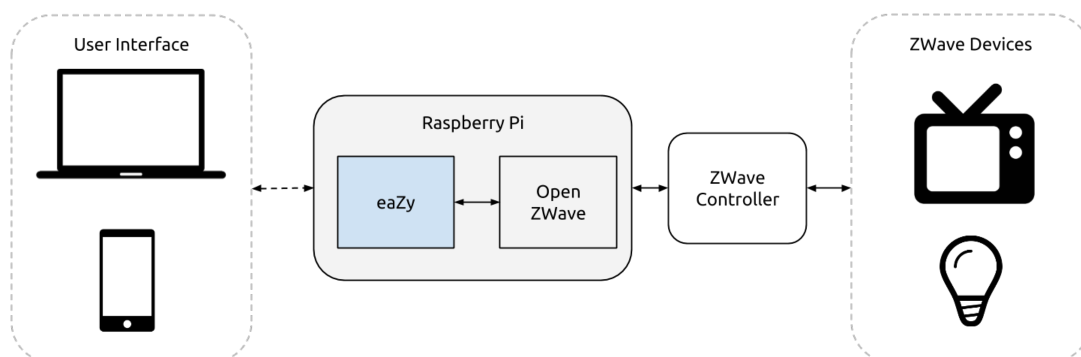


Figure 1: EAZY overview.

EAZY is a Home Automation system that aims on simple automation of a house, with which the user can access information about his home environment and take control of it even remotely. The solution must be able to communicate with *Zwave* devices to control lightning, appliances and sensors, and provide information about temperature, luminosity and energy consumption. Figure 1 shows an overview of the context of the system.

The EAZY key features are:

- Device Control: allows user to interact with the house devices to get their information, such as device consumption, and also change their state (*ie*. usually on or off).
- Scene management: the system must provide scene definitions. Given a predefined state of a device, some pre-configured action is triggered (*eg*. if movement is detected, the lights shall turn on).
- Flexible access: it is possible to access the system through any device that presents a web browser, simultaneously and in a responsive manner.

The solution is compatible with Linux operating system and does not require high performance hardware to operate, meaning it can be ran on portable computers such as the Raspberry Pi, shown in Figure 2. The user interaction is made through a *Web* interface, accessible via any browser.

Figure 2: Raspberry Pi Model B.

Source: http://www.raspberrypi.org/products/model-b/

3. eaZy Development

As technical requirements to the system, the keywords are **flexibility**, **scalability** and **simplicity**. Following these premises, the system is composed by three components:

- **eaZy Web**: component responsible for the User Interface, accessible via a web browser. Does not maintain any states or holds automation logic.
- **Home Stack**: responsible for the Home Automation logic. This is the central component of the system, which handles data persistency, scenes management and provides all information to the UI component. Also, Home Stack sends commands to and receives updates from the Device Controller.

- **Device Controller**: is the component responsible for the communication with the device network, that is, the interface between the system and the actual devices.

  Figure 3 shows the systems architecture and tech stack.

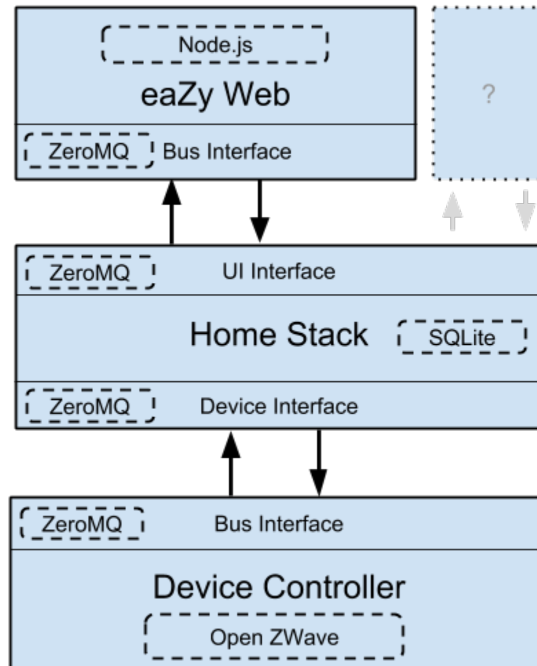


Figure 3: EAZY Architecture v3.

The communication between each component has a key role on the system functioning. This feature should reflect the flexibility requirement of the solution, and therefore was implemented with the following concerns: components are able to communicate with each other, no matter the language they are implemented in; a server-like component (*ie.* Home Stack) should be able to send messages to its client-like components (*ie.* user interface components), without actually implementing specific logic for each.

A pattern for this communication was implemented and externalized as a library, called Communication Bus. The Communication Bus was implemented using *ZeroMQ* [11], which is framework that enables communication between processes via a socket API abstraction.

4. Results

The main product generated by this work is the EAZY source code. The system is open source, MIT licensed. One can view and contribute to the project using the the tool *Github*, which is a collaboration platform that uses *Git* for version controlling projects. The address of the EAZY repository is: <`https://github.com/hoffmannmatheus/eaZy`>.

The Figures 4 and 5 show screenshots from the system, taken from a Linux desktop and an Android mobile device.

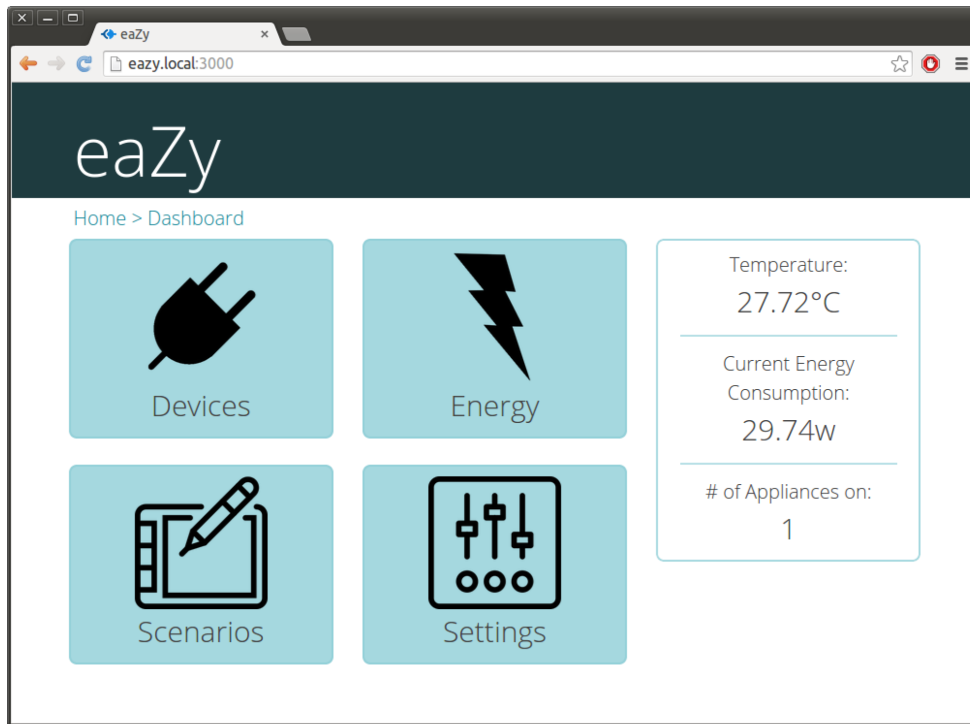
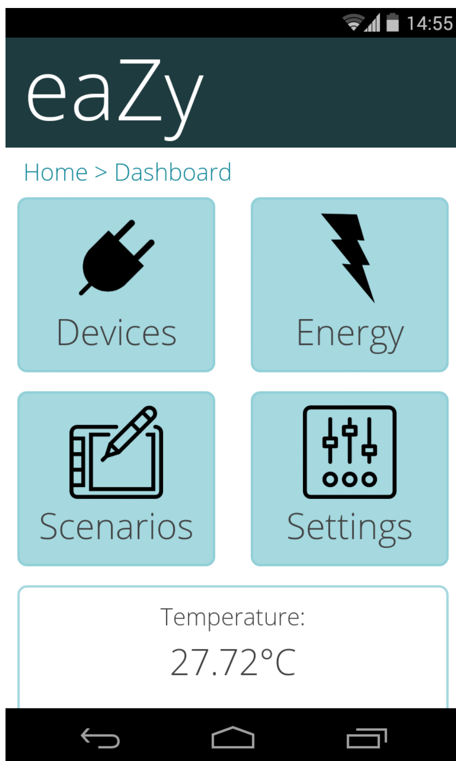


Figure 4: Desktop screenshot from the EAZY *Dashboard*.



(a) EAZY *Dashboard*.

(b) EAZY *Devices*.

Figure 5: EAZY screenshots from an Android mobile device.

5. Conclusion

Finally, the outcome of the project is a functional Home Automation system, ready to automate simple tasks in a real home environment. The resulting code is released as open source, and there are guidelines for possible users to setup EAZY in their homes.

As future works, there is a number of improvements and new features that can be implemented. For example:

- User control: there is currently no access control or login mechanism to track user activity and protect the network against malicious behavior.
- New Analytics Component: which would be a new component capable of generating interesting statistics about energy consumption, monetary expenses and device usage.

References

[1] HECKMAN, Davin. The Revolutions in Personal Computing and Origin of the Smart House. In: HECKMAN, Davin. **A Small World: Smart Houses and the Dream of the Perfect Day**. Los Angeles: Duke University Press, 2008. p. 55-56.

[2] DENNIS, Andrew K. What home automation is. In: DENNIS, Andrew K. **Raspberry Pi Home Automation with Arduino**. Birmingham: Packt Publising, 2013. p. 17-19.

[3] HONEYWELL. **H316 General Porpouse Digital Computer**. Framingham: Honeywell, 1970.

[4] WEI, Chao-huang; THAN, Hoang; WANG, Yu-ning. **Realization of Home Appliances Control System based on Power Line Communication Technology**. Taiwan: Southern Taiwan University, 2012.

[5] SILVA, Matheus Hoffmann. **Sistema de Automação Residencial baseado em Raspberry Pi e Open ZWave**. 2014. 96 f. Thesis (Bachelor) - Information Systems, Federal University of Santa Catarina, Florianópolis, 2014.

[6] INSTEON. **Insteon Whitepaper: The Details**, Version 2.0. Irvine: INSTEON, 2013.

[7] KINNEY, Patrick. **ZigBee Technology: Wireless Control that Simply Works**. Communications Design Conference, 2013.

[8] ZWave Alliance. **About ZWave Technology**. Disponível em: <http://www.z-wavealliance.org/about-z-wave>. Access in: 30 jun. 2014.

[9] PULSE Work. **UPB Technology Descriptio**n, Version 1.4. Northridge: Pulse Work, 2005.

[10] OpenZWave Community. **OpenZWave Library**. Disponível em: <http://www.openzwave.com/dev/>. Access in: 30 jun. 2014.

[11] HINTJENS, Pieter. **ZeroMQ Code Connected**. iMatix Corporation and Contributors. Disponível em: <http://zguide.zeromq.org/>. Access in: 15 Set. 2014.