UNIVERSIDADE FEDERAL DO PARANÁ

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ANDREY VIKTOR KANDAUROFF

DIEGO LUIZ MOLINA

SISTEMA IOT DE AUTOMAÇÃO RESIDENCIAL VIA CLOUD E DISPOSITIVOS MÓVEIS

CURITIBA

2019

ANDREY VIKTOR KAUDAROFF

DIEGO LUIZ MOLINA

SISTEMA IOT DE AUTOMAÇÃO RESIDENCIAL VIA CLOUD E DISPOSITIVOS MÓVEIS

Trabalho apresentado como nota parcial à disciplina de Trabalho de Conclusão de Curso II de Engenharia Elétrica da Universidade Federal do Paraná.

Orientador: Prof. Dr. Marcos Vinicio Haas Rambo

CURITIBA

2019

SUMÁRIO

LISTA DE FIGURAS

LISTAS DE TABELAS

LISTAS DE SIGLAS

ABNT	Associação Brasileira de Normas Técnicas.
ASCII	American Standard Code for Information Interchange.
A/D	Analógico-Digital.
Bit	Binary digit.
Byte	Unidade de armazenamento.
C	Linguagem de programação.
CPU	Unidade Central de Processamento.
CRC	Cyclic redundancy check.
FIFO	First In, First Out.
FTDI	Future Technology Devices International.
FTP	File Transfer Protocol.
HTTP	Hypertext Transfer Protocol
IDE	integrated development environment.
IER	Interrupt Enable Register.
I/O	Entrada e saída.
IP	Internet Protocol.
Java	Linguagem de Programação.
OSI	Open Systems Interconnection.
PAD	Película de Cobre.
PC	Personal Computer.
PLC	Power Line Communication.
PWM	Pulse Width Modulation.
RBR	Receiver Buffer Register.
RO	Read Only.
SMD	Surface Mounted Device.
SQL	Structured Query Language.
SSH	Protocolo de Rede Secure Shell.
TCP	Transmission Control Protocol.
THR	Transmiter Holding Register.
TTL	Transistor-Transistor Logic.
UART	Universal Asynchronous Receiver/Transmitter.
UDP	User Datagram Protocol.

X-10	Padrão de comunicação via rede elétrica.
XML	EXtensible Markup Language.
Wi-Fi	Wireless Fidelity.
WO	Write Only.

1. INTRODUÇÃO

Automação residencial define-se pelo uso da tecnologia para facilitar e tornar automáticas tarefas habituais que em uma casa convencional ficaria a cargo de seus moradores. Com sensores de presença, temporizadores ou mesmo com um simples botão do controle remoto é possível acionar cenas ou tarefas pré-programadas, trazendo maior praticidade, segurança, economia e conforto para seus moradores (AUTOMATICHOUSE, 2016).

Por muitos anos, desde que o francês Júlio Verne introduziu o tema da automação para os livros de literatura que escreveu, a automação residencial tem sido um tema característico da literatura de ficção científica. Porém, ela só começou a se tornar algo praticável, a partir da introdução generalizada de eletricidade para a casa, no início do século XX. Notadamente com o fim da primeira grande guerra e, concomitantemente com o gradual avanço dos sistemas de controle automático e da tecnologia da informação é que ela vem se desenvolvendo, mesmo que tais inovações não sejam originalmente idealizadas para a aplicação em automação de casas, elas passam posteriormente a sê-lo (LENZ, 2012).

A década de 70 pode ser considerada o marco inicial da automação residencial, quando foram lançados nos EUA os primeiros módulos inteligentes chamados X-10. O protocolo X-10 utilizava a rede elétrica como canal de comunicação entre os diversos dispositivos de automação.

Mais adiante, na década de 80, com a popularização dos computadores pessoais (*PCs*), em detrimento aos *mainframes,* pôde-se pensar em um PC como uma central de automação. Entretanto, a grande desvantagem desse sistema é o elevado consumo, devido à necessidade de manter o *PC* sempre ligado. Outra desvantagem está na centralização do controle que pode vir a ser falho e comprometer o funcionamento de todo o sistema automatizado. A partir desses problemas parte-se para o desenvolvimento de dispositivos dedicados através da utilização de microprocessadores e microcontroladores e da exclusão dos *PCs* (BORTOLUZZI, 2013).

A internet banda larga concedeu ao usuário a possibilidade de controle e monitoramento da residência de qualquer lugar que disponha do serviço. Acrescenta-se a este fato a convergência tecnológica intensificada a partir do século

XXI, na qual um mesmo dispositivo (celular, *smarphone, tablet*, etc.) pode incorporar diferentes serviços (telefonia, internet, monitoramento, controle da residência e assim por diante). Nesses casos, um *software* aplicativo realiza controle das automações (BORTOLUZZI, 2013).

1.1. OBJETIVOS

Os objetivos desse trabalho são apresentados a seguir, de forma a apresentar os objetivos geral e específico separadamente.

1.1.1. Objetivo Geral

Este projeto tem por objetivo geral o desenvolvimento de um sistema de automação residencial utilizando computação em nuvem que seja controlado por um dispositivo móvel.

O sistema do TCC será composto por três módulos independentes e descentralizados:

- Módulo de controle e monitoramento de tomadas;
- Módulo de segurança de vazamento de gás e detecção de fumaça;
- Projeto e módulo de um sistema de controle de iluminação.

1.1.2. Objetivos específicos

Dentre os principais objetivos específicos destacam-se:

- Desenvolver um software para *smartphones* que faça a interface do usuário com o sistema de maneira inteligente baseado no conceito de Internet das Coisas.
- Elaborar e desenvolver os hardwares e atuadores do sistema (circuitos de iluminação, tomadas e sensores de gás e fumaça).
- Estudar e implementar o protocolo de comunicação MQTT (Message Queuing Telemetry Transport) em nuvem com o gateway do sistema.
- Realizar a comunicação entre os dispositivos do sistema: *gateway*, nuvem, atuadores e dispositivo móvel de controle.

2. FUNDAMENTAÇÃO TEÓRICA

Este projeto envolve alguns conceitos fundamentais teóricos que devem ser compreendido para o total entendimento do trabalho, sendo eles a respeito de redes de computadores, *sockets*, microcontroladores, linguagem de programação, eletrônica e comunicação *peer-to-peer*.

Para a comunicação entre o usuário e o sistema será utilizado um *software* desenvolvido em linguagem de programação orientada a objeto Java para aparelhos *smartphones* com sistema operacional Android 4.0 ou superior. Já a comunicação entre o *software* e o controlador será feita através de um módulo *Wi-Fi* via protocolo *MQTT (Message Queuing Telemetry Transport).*

A FIGURA 1 apresenta uma visão geral e genérica do projeto em formato de diagrama de blocos.

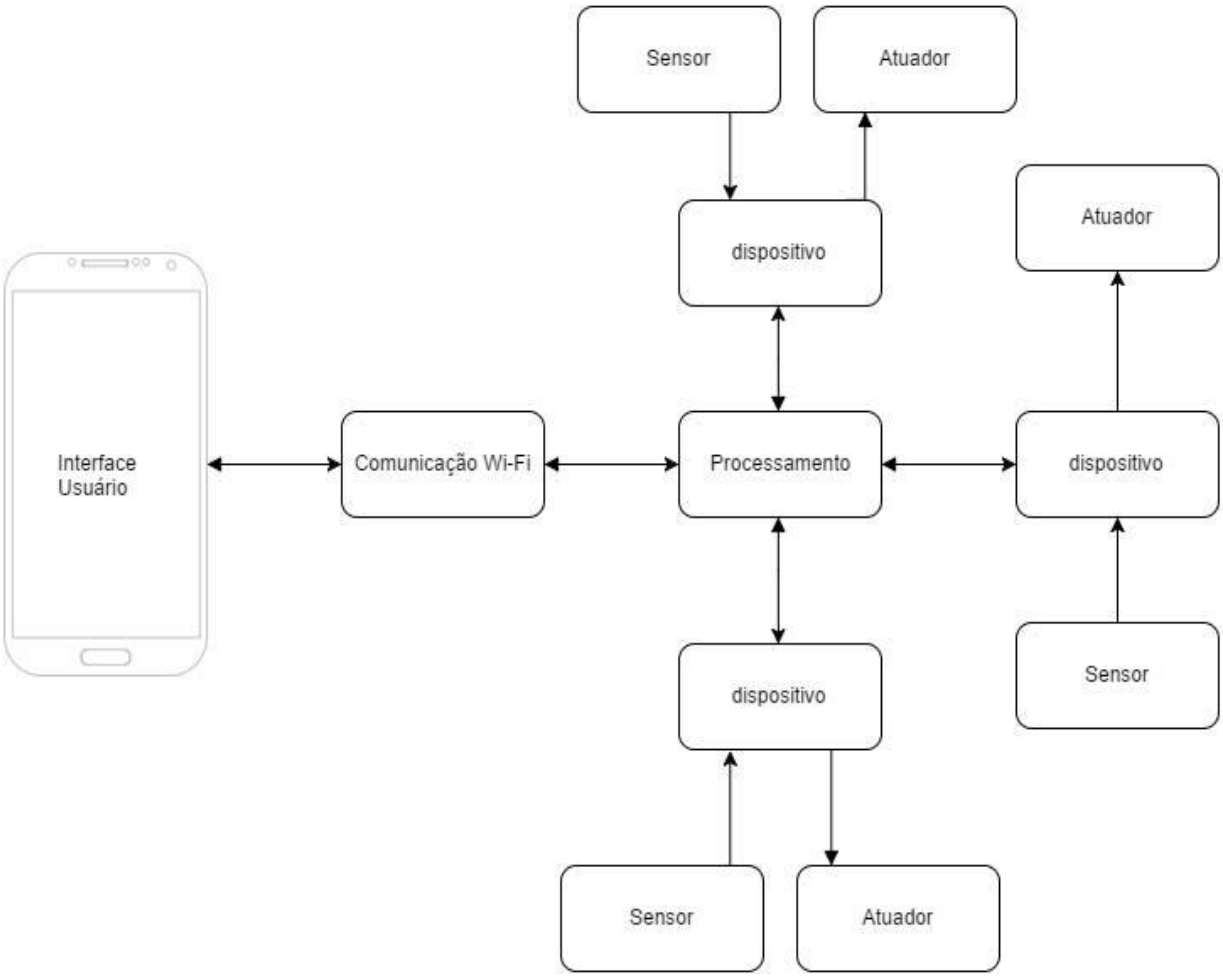

FIGURA 1. DIAGRAMA DE BLOCOS GENÉRICO DO SISTEMA

FONTE: AUTOR (2018)

Para se concretizar a execução do objetivo principal do projeto, foram estabelecidos objetivos secundários, sendo eles: análise da tecnologia utilizada em automação doméstica no Brasil; estudos de redes de computadores e internet; estudos sobre diferentes tipos de comunicação; estudo a respeito de desenvolvimento de *software* para *smartphones* em Java e sistema embarcado Android; funcionamento de módulos *Wi-Fi* e controladores; dimensionamento e *layout* de placas.

.

2.1. REDE DE COMPUTADORES

Uma rede de computadores (FIGURA 2) é formada por um conjunto de módulos processadores capazes de trocar informações e compartilhar recursos, conectados por um sistema de comunicação (meios de transmissão e protocolos) (NASCIMENTO, 2012).

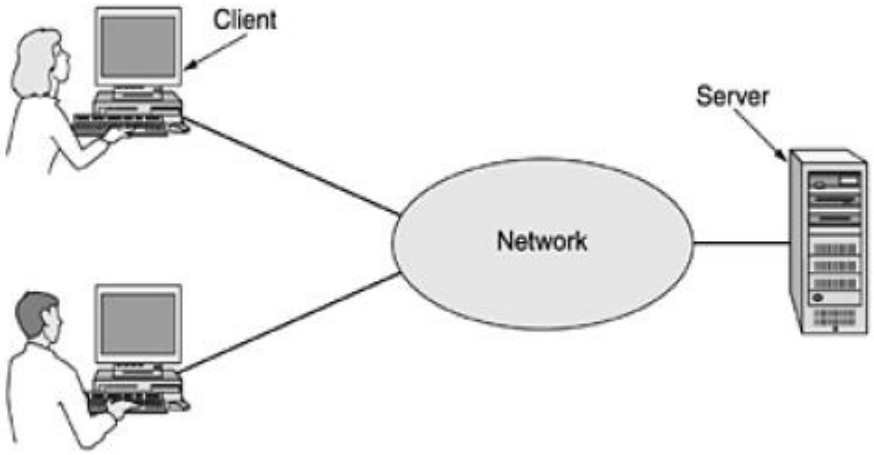

FIGURA 2. MODELO CLIENTE/SERVIDOR
FONTE: TANENBAUM (2011).

As redes de computadores são organizadas e separadas em níveis, como uma pilha de camadas uma sobreposta a outra. Cada camada tem a sua função definida para a camada superior, no entanto ela é invisível para as camadas inferiores que não convêm sua função.

Existe um protocolo conhecido como protocolo da camada n, sua função é ditar regras e convenções a respeito da comunicação entre camadas do mesmo nível, onde, por exemplo, a camada n de uma máquina se comunica com a camada n de outra máquina, conforme exposto na FIGURA 3.

A sétima camada do modelo *OSI* é conhecida como a camada de aplicação e é onde são implementados os *sockets,* ou seja, as camadas inferiores são invisíveis para quem utiliza. As únicas mudanças que podem ser feitas nelas são as portas utilizadas para as conexões e o tipo escolhido de protocolo na camada de transporte.

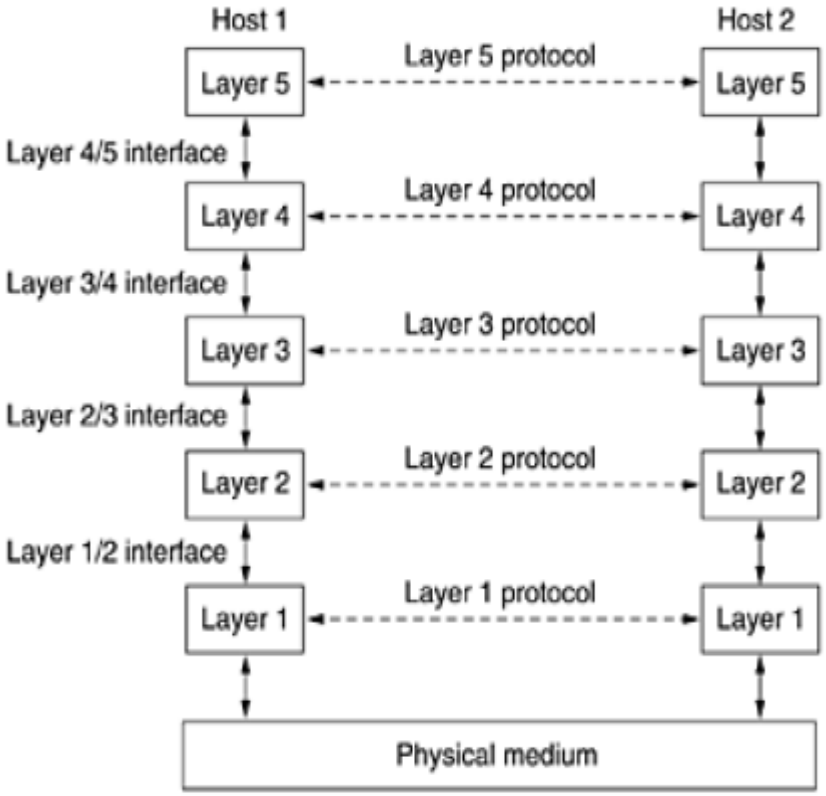

FIGURA 3. CAMADAS, PROTOCOLOS E INTERFACES
FONTE: TANENBAUM (2011).

Socket é um mecanismo de comunicação, usado normalmente para implementar um modelo cliente/servidor, que permite a troca de mensagens (FIGURA 4) entre os processos de uma máquina/aplicação servidor e de uma máquina/aplicação cliente (PINTO, 2012). *Socket* tem a função de efetuar numa determinada função às tarefas de redes. Existem dois principais tipos de *socket* o *TCP* e o *UDP*.

2.2. *SOCKETS TCP*

O *TCP* é um protocolo de rede que fornece um transporte *full-duplex* baseado em conexão. Este protocolo também oferece fragmentações e remontagens de mensagens e suporta mensagens de qualquer tamanho que tenha vindo de camadas superiores. O *TCP* pode realizar fragmentações dos fluxos de mensagens em segmentos que possam ser manuseados pelo *IP*.

A versatilidade e a robustez desse protocolo tornou-o adequado às redes globais, já que ele verifica se os dados são enviados de forma correta, na sequência adequada e também sem erros, pela rede (MARTINS, 2010). A maioria das aplicações cibernéticas são aplicadas sobre o *TCP*, como o *SSH*, *FTP*, *HTTP*.

Outras características do protocolo *TCP* são:

- Orientado à conexão;
- Ponto a ponto;
- Confiabilidade;
- *Handshake;*
- Entrega ordenada;
- Controle de fluxo;

2.3. *SOCKET UDP*

O *UDP* é um protocolo feito para transmitir dados poucos sensíveis como uma *streaming* de áudio de vídeo, por exemplo. No *UDP* não existe verificação de que os dados do pacote tenham sido enviados ou recebidos com sucesso. Os dados são transmitidos apenas uma vez, incluindo apenas um sistema de *CRC*. Os pacotes que chegam danificados são simplesmente descartados, sem que o emissor sequer saiba do problema. A ideia é justamente transmitir dados com o maior desempenho possível, eliminando dos pacotes tudo que não seja informação necessária. Apesar da pressa, o *UDP* tem seus méritos, afinal não é interessante que quadros fantasmas apareçam no meio de um vídeo, muito menos se isso ainda por cima causasse uma considerável perda de desempenho (MARTINS, 2010). Em geral, programas que utilizam protocolo *UDP* recorrem também a uma porta *TCP* para

enviar requisições de dados a serem enviados e também para verificar constantemente se o cliente ainda esta online (MARTINS, 2010).

As vantagens do protocolo *UDP* são:

- O *UDP* é uma opção adequada para fluxos de dados instantâneos;
- O *UDP* também comporta *broadcasting* e *multicasting*;
- O *UDP* não perde tempo com criação ou destruição de conexões. Durante uma conexão o *UDP* troca apenas 2 pacotes, diferente do *TCP* que esse número é superior a 10.

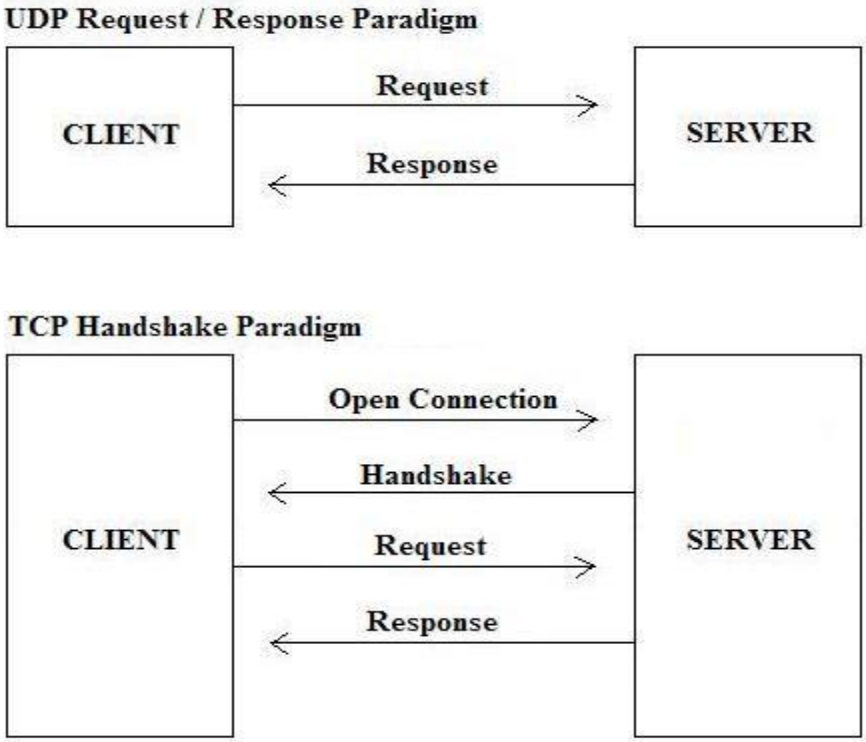

FIGURA 4. *UDP* x *TCP* MODELOS DE REQUISIÇÃO/RESPOSTA
FONTE: NATIONAL INSTRUMENTS (2011).

2.4. MICROCONTROLADORES

Um microcontrolador é um sistema computacional completo, no qual estão incluídos uma *CPU*, memória de dados e programa, um sistema de *clock*, portas de *I/O*, além de outros possíveis periféricos (FIGURA 5), tais como, módulos de temporização e conversores *A/D* entre outros, integrados em um mesmo componente (DENARDIN, 2011). As partes integrantes de qualquer computador, e que também estão presentes, em menor escala, nos microcontroladores são:

- Unidade Central de Processamento;
- Sistema de *clock* para dar sequência às atividades da *CPU*;
- Memória para armazenamento de instruções e para manipulação de dados;
- Entradas para interiorizar na *CPU* informações do mundo externo;
- Saídas para exteriorizar informações processadas pela *CPU* para o mundo externo;
- *Firmware* para definir um objetivo ao sistema.

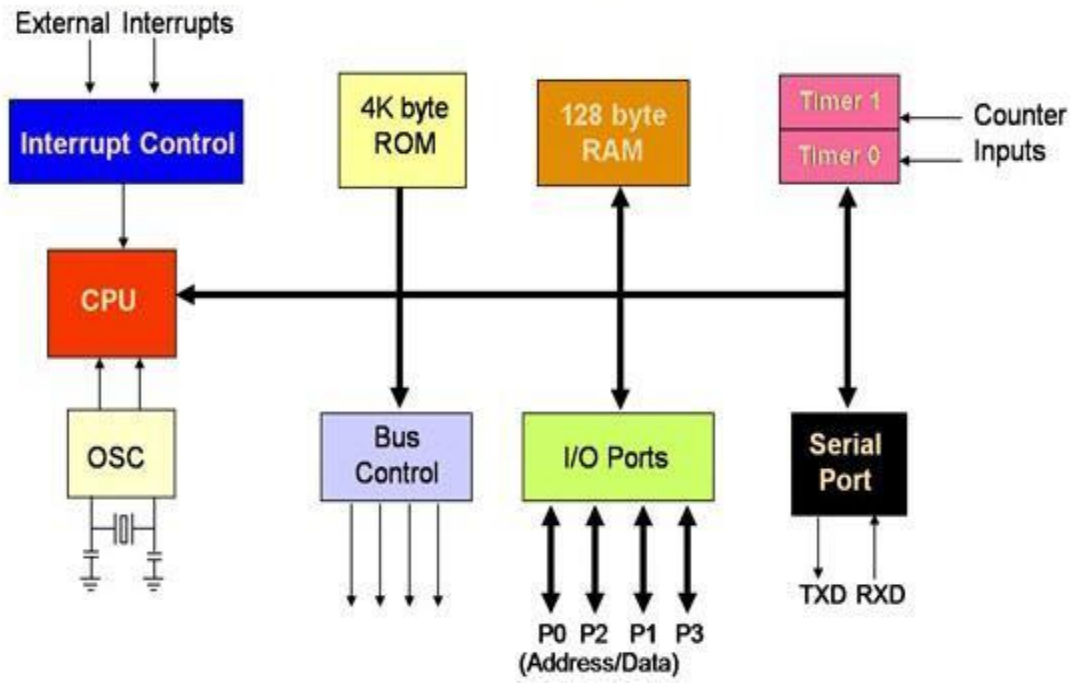

FIGURA 5. DIAGRAMA DE BLOCOS DO MICROCONTROLADOR 8051
FONTE: TUTORIALS POINT.

2.4.1. Arquitetura de Von-Neumann

Na arquitetura Von-Neumann (FIGURA 6), os barramentos de dados e endereços são compartilhados entre memórias de programas e memórias de dados na comunicação com a *CPU*. Nesse tipo de arquitetura, quando a *CPU* está acessando a memória de programa ela não consegue acessar a memória de dados, porque usa os mesmos barramentos para as duas memórias (CAVALCANTE, 2012).

A separação entre a *CPU* e a memória leva para o gargalo de Von-Neumann, a produção limitada (taxa de transferência) entre a *CPU* e a memória em

comparação com a quantidade de memória. Na maioria dos computadores modernos, o *throughput* é muito menor do que a taxa com que o processador pode trabalhar. Isso limita seriamente a velocidade de processamento eficaz quando o processador é exigido para realizar o processamento mínimo em grandes quantidades de dados (CAVALCANTE, 2012).

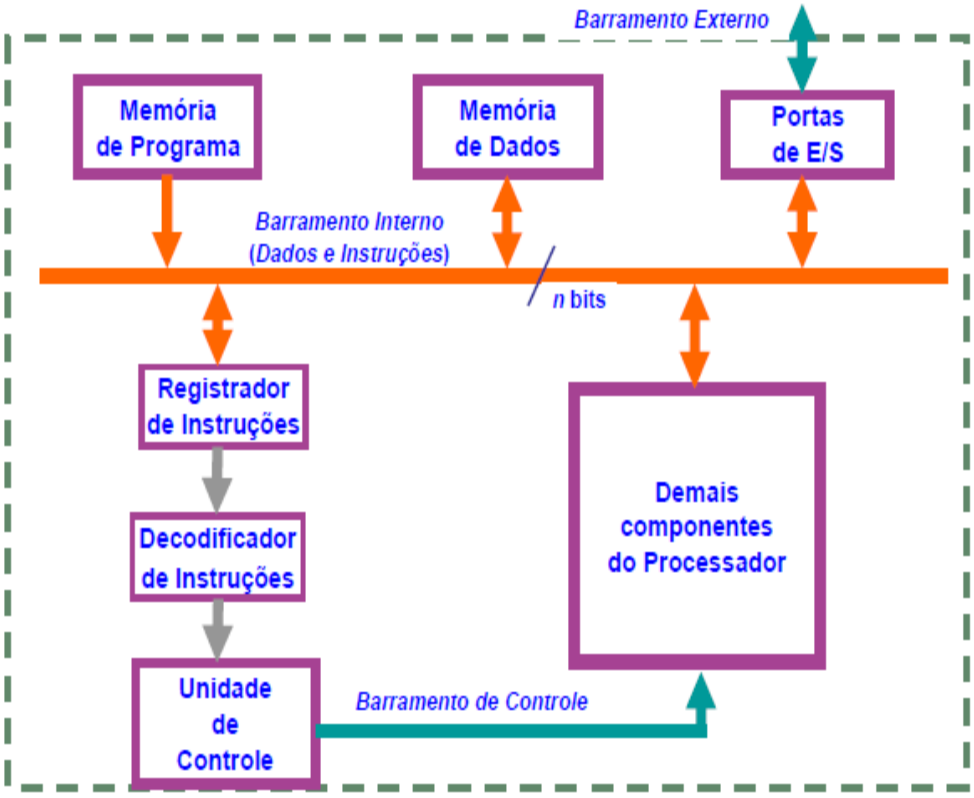

FIGURA 6. DIAGRAMA ARQUITETURA DE VON-NEUMANN
FONTE: PASTRO (2014).

2.4.2. Arquitetura de Harvard

A arquitetura de computadores *Harvard* (FIGURA 7) se diferencia das outras por possuir duas memórias distintas e independentes em termo de barramento e conexão com o processador, isso possibilita acessar a memória de dados separadamente da memória de programa.

A principal vantagem dessa arquitetura é que a leitura de instruções e de alguns tipos de operandos pode ser realizada ao mesmo tempo em que a execução das instruções (tempo *Tcy*). Isso significa que o sistema fica o tempo todo executando instruções, o que acarreta um significativo ganho de velocidade.

Enquanto uma instrução está sendo executada, a próxima está sendo lida. Esse processo é conhecido como *pipelining* (canalização) (CAVALCANTE, 2012).

Este tipo de arquitetura possui um repertório com menos instruções que a do Von-Neumann, e essas são executadas em um único ciclo de *clock*.

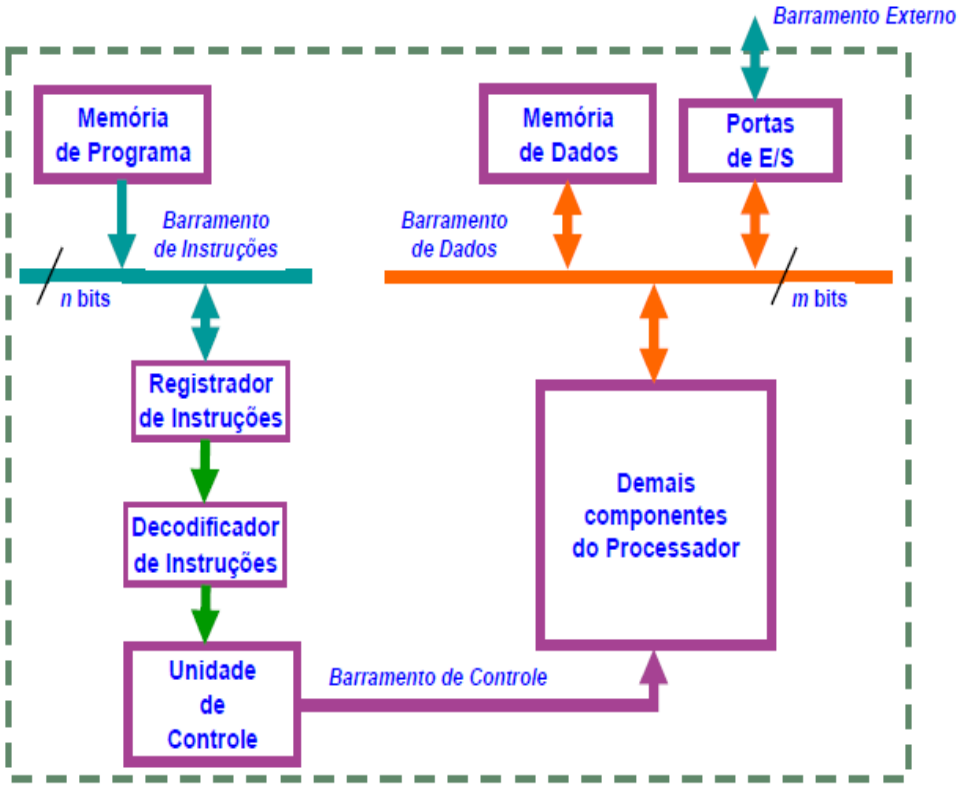

FIGURA 7. DIAGRAMA ARQUITETURA DE HARVARD
FONTE: PASTRO (2014).

2.4.3. *CISC – Complex Instruction Set Computer*

Este tipo de computador denomina-se por ser um computador com *set* de instrução mais complexo e quanto maior a complexidade da instrução que será executada, mais espaço ela ocupará no *chip*, podendo chegar um momento em que a instrução seja tão grande que começará a afetar o desempenho do sistema, afetando assim a implementação de outras funções importantes (PEDROSO, 2016).

Numa análise feita pelo laboratório da IBM sobre como estavam sendo usados os diversos tipos de instruções, concluíram que num microprocessador que usava um set de instruções de, por exemplo, 200 instruções, a maior parte do processamento era feita apenas com 10 instruções em média (CAVALCANTE,

2012). Então para se executar todas as instruções é necessário um ou mais ciclos de máquinas. A FIGURA 8 apresenta um diagrama esquemático da arquitetura *CISC*.

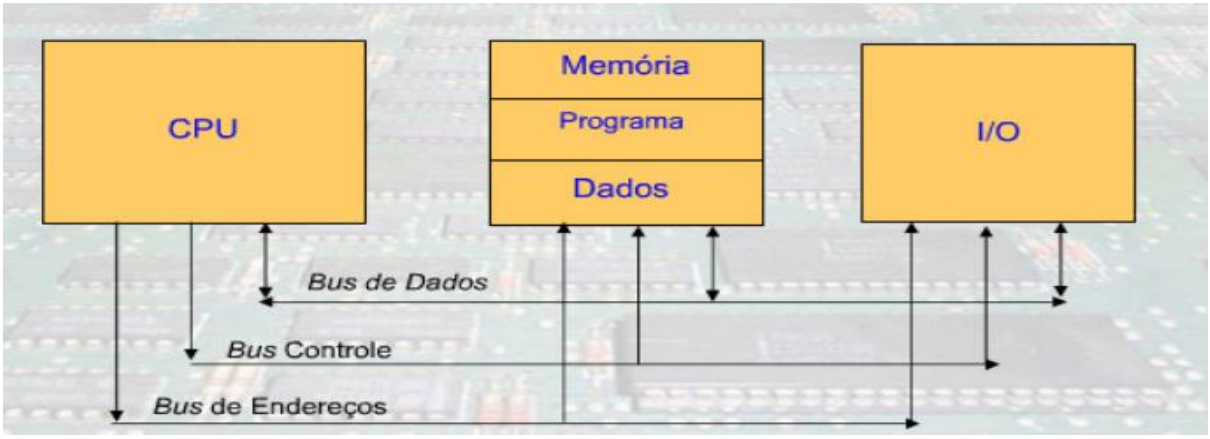

FIGURA 8. DIAGRAMA ARQUITETURA *CISC*
FONTE: CAVALCANTE (2012).

2.4.4. *RISC – Reduced Instruction Set Computer*

Computador do tipo *RISC* é denominado como um computador com um conjunto reduzido de instruções. Esta arquitetura de processadores favorece um conjunto simples e pequenas que são executadas no mesmo ciclo de máquina, isso faz com que o processador execute várias instruções ao mesmo tempo (isso é possível devido ao *pipeline),* tornando o processamento muito mais rápido (PEDROSO, 2016). A FIGURA 9 apresenta o diagrama esquemático da arquitetura *RISC* de computadores.

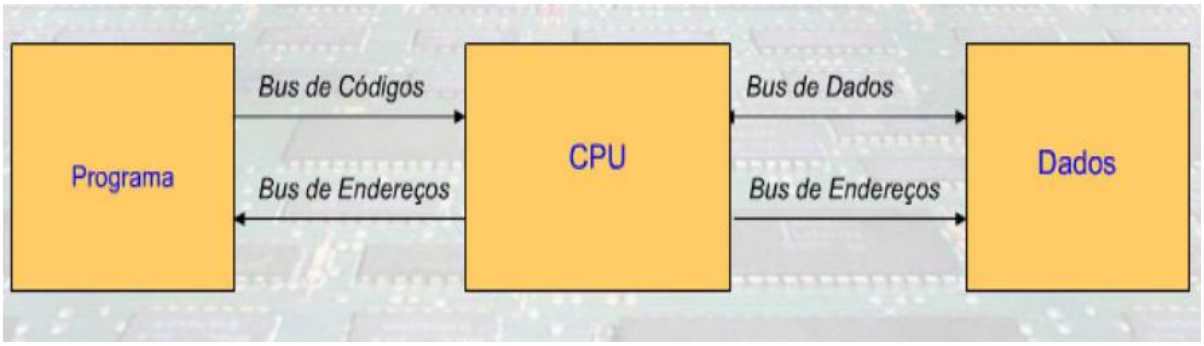

FIGURA 9. DIAGRAMA ARQUITETURA *RISC*
FONTE: CAVALCANTE (2012).

2.5. LINGUAGEM DE PROGRAMAÇÃO

Uma linguagem de programação é um conjunto de códigos ou regras sintáticas e semânticas padronizadas que tem por objetivo expressar uma determinada instrução para um computador (TARCÍSIO, 2014).

2.5.1. Programação Estruturada

Esta forma de programação possui três estruturas básicas de controle (FIGURA 10): sequência, decisão ou condição e iteração. “A programação estruturada, quando bem feita, possui um desempenho superior ao que se pode ver na programação orientada a objetos, isso ocorre pelo fato de ser um paradigma sequencial, em que cada linha de código é executada após a outra, sem muitos desvios, como vemos na *POO*” (MACHADO, 2015).

A linguagem de programação em C é a principal representante da programação estruturada e sua principal utilização, devido ao baixo nível de programação, é para sistemas embarcados ou em outros em que o conhecimento do *hardware* se faz importante para um bom programa (MACHADO, 2015).

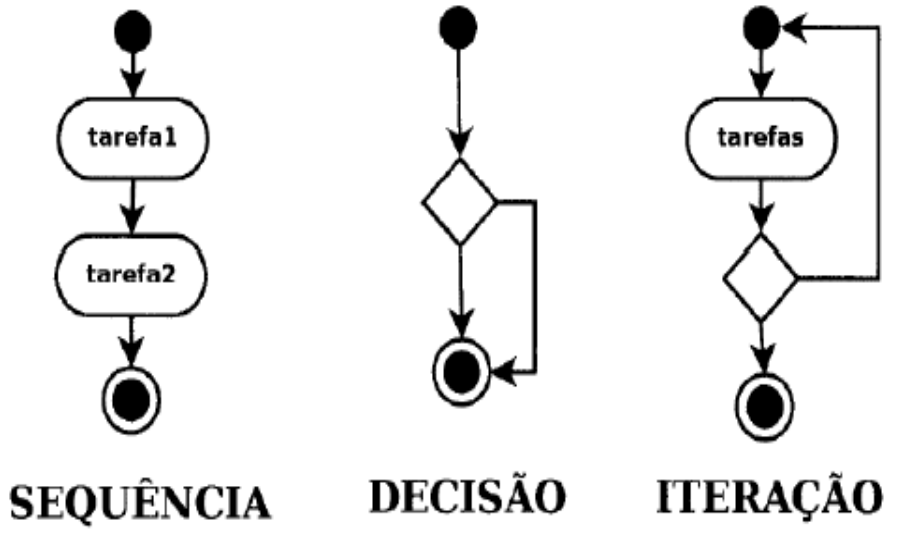

FIGURA 10. DIAGRAMA PROGRAMAÇÃO ESTRUTURADA
FONTE: MARINHO (2013).

2.5.2. Programação Orientada a Objetos

A programação orientada a objeto segue um caminho de desenvolvimento que é seguido por algumas linguagens de programação, entre elas o C# e Java (MACHADO, 2015).

Esta forma de desenvolvimento de *software* possui alguns paradigmas principais que se fundamentam na utilização de componentes individuais (objetos) que colaboram para construir um sistema mais complexo. São eles: classes, objetos, associação, encapsulamento, herança e polimorfismo. A programação é feita por meio da definição de classes e criação de hierarquias, nas quais propriedades comuns são transmitidas das superclasses para as subclasses através do mecanismo de herança (SOUZA, 2015).

Uma classe é uma maneira de definir um tipo de dado em linguagem orientada a objeto, ela é formada por dados e comportamentos. Objetos dessa classe são instanciados de tal maneira que a compilação do programa é vista como um conjunto de objetos relacionados que se comunicam enviando mensagens uns para os outros.

Um objeto informalmente representa uma entidade física, conceitual ou de *software.* Também é possível afirmar que é um conceito, uma abstração, ou entidades com limites bem definidos e um significado para a aplicação. “Os dados de um objeto são totalmente escondidos e protegidos de outros objetos. A única maneira de acessá-los é através da invocação de uma operação declarada na interface pública do objeto. A interface pública de um objeto consiste no conjunto de operações que um cliente do objeto pode acessar” (SOUZA, 2015).

2.6. *SOFTWARE* ANDROID

O desenvolvimento de *software* para Android é o processo pelo qual um novo aplicativo é criado para o sistema operacional Android, o qual é muito utilizando em *smartphones* e *tablets*. Aplicativos são geralmente desenvolvidos na linguagem de programação Java usando ambientes de desenvolvimento.

2.6.1. Android Views

As Android Views são os principais componentes utilizados para desenvolver as interfaces dos aplicativos para Android. Além de serem responsáveis pela maior parte da interação entre o usuário e o aplicativo, elas têm uma grande responsabilidade em definir o design da interface para que seja amigável com uma alta usabilidade.

Com exceção de alguns casos, todos os aplicativos desenvolvidos têm alguma forma de interface com usuário. No Android, isso é feito através do uso de uma hieraquia de Views e ViewGroups (FIGURA 11).

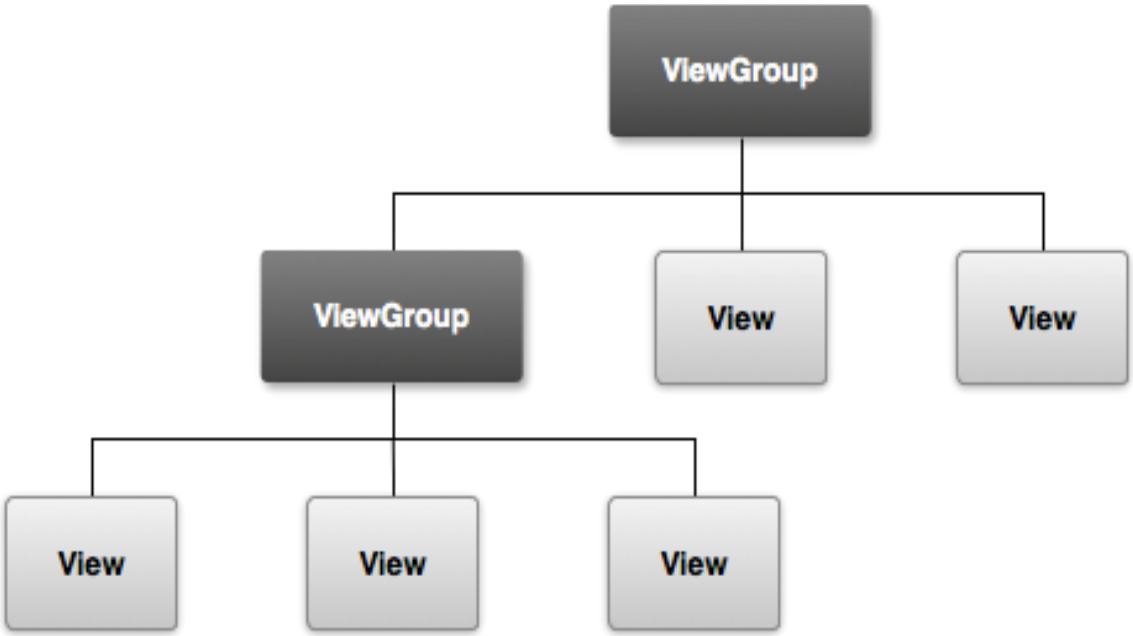

FIGURA 11. HIERARQUIA VIEW E VIEWGROUP
FONTE: FILLIPE CORDEIRO (2016).

Uma *View* no Android pode ser uma imagem, um pedaço de texto, um botão ou qualquer outra coisa que o aplicativo pode exibir. E todas as *Views* juntas formam o layout da interface (FIGURA 12).

Tudo que é visível e possível de interagir no aplicativo é chamado de interface de usuário ou UI (*User Interface*).

FIGURA 12. LAYOUT INTERFACE APLICATIVO ANDROID
FONTE: FILLIPE CORDEIRO (2016).

Cada *software* para o Android pode ser dividido em Views individuais que o compõem (FIGURA 13). Ou seja, as Views são os componentes básicos usados para construir o layout do aplicativo.

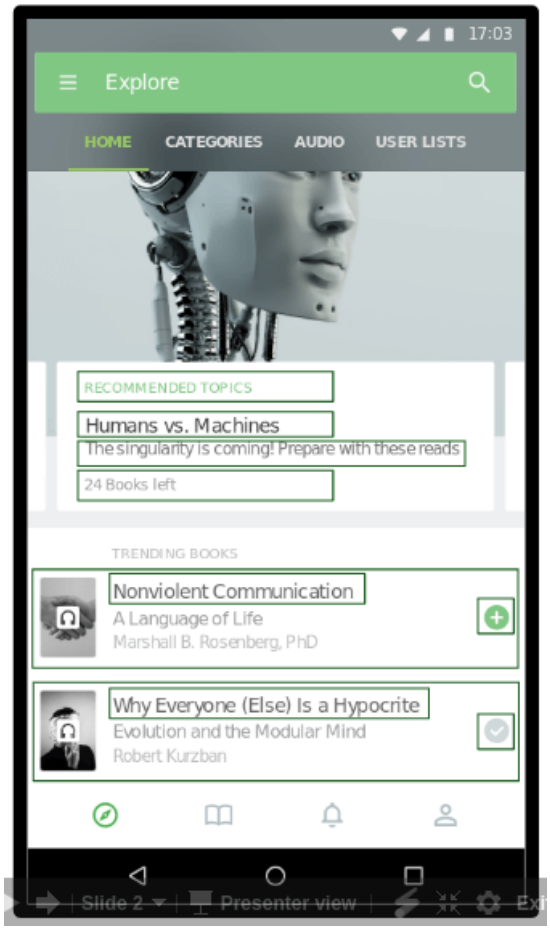

FIGURA 13. VIEWS DESTACADOS NO LAYOUT INTERFACE APLICATIVO ANDROID
FONTE: FILLIPE CORDEIRO (2016).

Para declarar o *layout*, é possível instanciar objetos *View* no código e começar a criar uma árvore. Mas a forma mais comum e efetiva de definir o *layout* é com um arquivo XML. O XML oferece uma estrutura legível por humanos para o *layout*, similar a HTML. Por exemplo, um texto pode ser implementado utilizando o *TextView* (FIGURA 14).

```
<TextView
    android:text="Olá Androideiro!"
    android:background="@android:color/darker_gray"
    android:layout_width="150dp"
    android:layout_height="75dp" />
```

FIGURA 14. IMPLEMENTAÇÃO DE TEXTO UTILIZANDO TEXTVIEW
FONTE: FILLIPE CORDEIRO (2016).

O *XML* deve sempre começar abrindo o elemento (*tag*) através da utilização de um (<), seguido do nome do componente. Este exemplo poderia ser uma *ImageView*, *Button*, ou qualquer outra *View*. Em seguida, implementa-se uma lista de atributos, um por linha e no final tem-se (/>) fechando nossa *tag*. Ou também é possível implementar uma *tag* separada para o fechamento (FIGURA 15), em um *LinearLayout* é possível a utilização.

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">

    <TextView
        android:text="Ola Androideiro!"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="24sp"  />

    <TextView
        android:text="Bora criar apps"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="24sp"  />

</LinearLayout>
```

FIGURA 15. IMPLEMENTAÇÃO DE TEXTO UTILIZANDO TEXTVIEW
FONTE: FILLIPE CORDEIRO (2016).

No código apresentado na FIGURA 15, percebe-se que no meio do *LinerarLayout* tem-se dois elementos *TextView*. Estes são chamados de elementos filhos dentro do *LinearLayout* pai.

Outra característica dos elementos *XML* das *Views*, são os textos adicionais dentro da *tag* que são chamados de atributos. Os atributos são características que determinam o comportamento ou a aparência da *View* no Android (CORDEIRO, 2016).

Analisando os atributos, observa-se que o nome do atributo fica do lado esquerdo e do lado direito tem-se o seu valor. É importante notar que os valores devem ser colocados entre aspas como parte da sintaxe do *XML*.

Para desenvolver um aplicativo Android, uma dificuldade bastante comum é criar *layouts* que se encaixem nos vários tamanhos e resoluções diferentes de telas de dispositivos presentes no mercado. É importante que todo aplicativo suporte o máximo de tamanho de telas diferentes para que funcione em vários aparelhos distintos.

A tela do *smartphone* é composta por pequenos quadrados chamados de *Pixels*. A FIGURA 16 apresenta uma *View* (um botão, por exemplo) com o tamanho de 2 *pixels* de altura por 2 *pixels* de largura para diferentes resoluções.

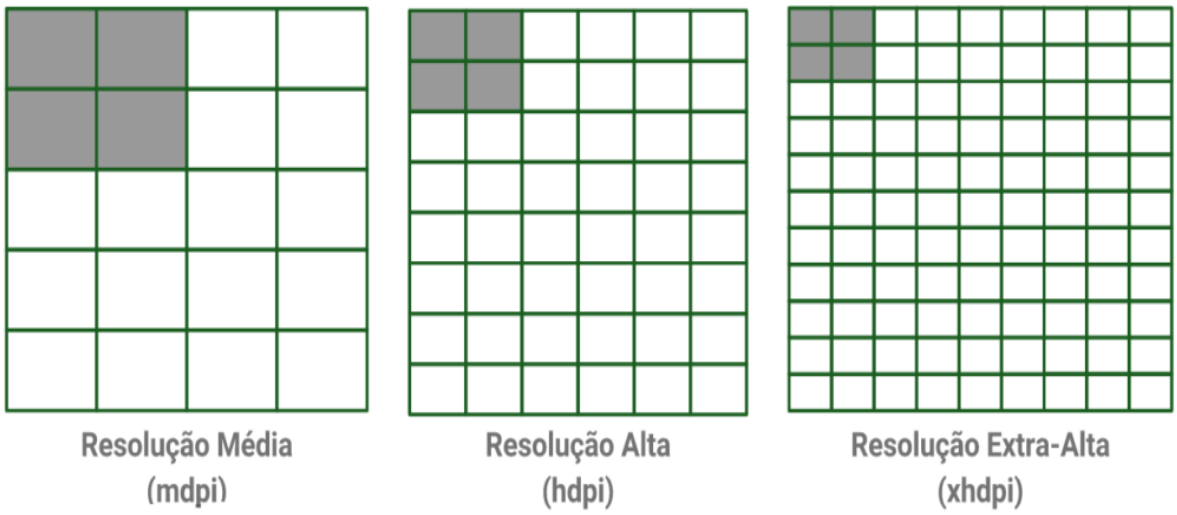

FIGURA 16. REPRESENTAÇÃO DO TAMANHO DO BOTÃO EM PIXELS
FONTE: CORDEIRO (2016).

É possível observar que na resolução *xhdpi* o usuário pode encontrar dificuldades para pressionar o botão. Para situações assim existe a medida *Pixels*

Independente da Densidade (DP), onde é possível definir o tamanho do botão por 2 DPs de altura por 2 DPs de largura (FIGURA 17).

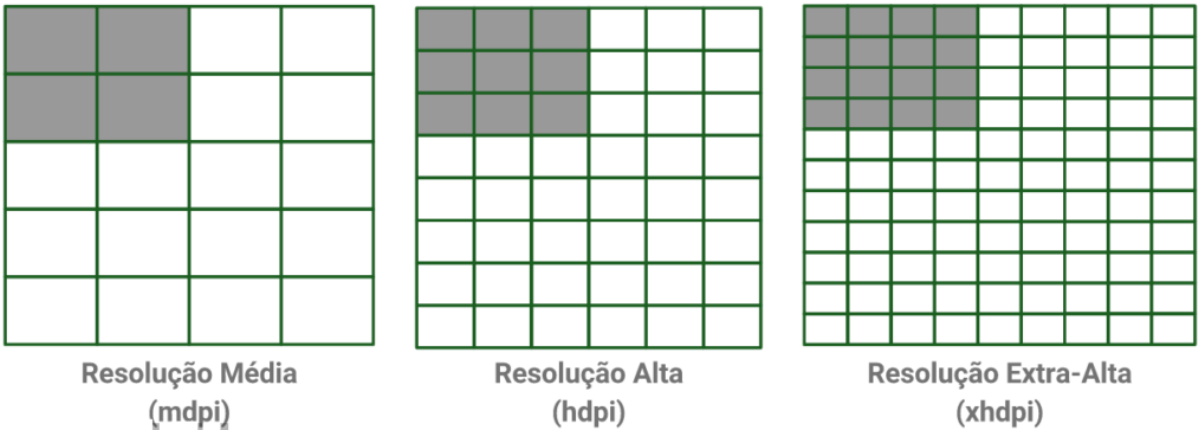

FIGURA 17. TAMANHO DO BOTÃO REPRESENTADO POR DP.
FONTE: CORDEIRO (2016).

Outro fator importante na hora de desenvolver um *software* para Android é definir quais *Views* utilizar no *layout*. Existem 6 opções (juntamente com seus *ViewsGroups*) que permitem desenvolver bons aplicativos, são elas:

- *TextView* – Exibe um de texto formatado;
- *ImageView* – Exibe um uma imagem;
- *Button* – É utilizado para e executar uma ação ao ser clicado;
- *ImageButton* – Exibe uma imagem com comportamento de um botão;
- *EditText* – É um campo de texto editável para a entrada de dados;
- *ListView* – É uma lista de itens que contém outras *Views*.

A utilidade do *TextView* é exibir o texto na tela de um aplicativo Android. A classe *TextView* contém uma lógica complexa que lhe permite exibir um texto formatado, *hyperlinks*, números de telefone, e-mails e outras funcionalidades úteis.

O *ImageView* é projetado especificamente para exibir imagens na tela. Isso pode ser usado para a exibição de recursos armazenados no aplicativo ou para a exibição de imagens que são baixadas da internet.

O *Button* é um dos controles mais básicos disponíveis em um aplicativo. Ele responde a cliques do usuário e chama um método em seu código para que você possa responder de forma apropriada quando o usuário pressiona o botão.

O *ImageButton* é um combinado de um Button com uma *ImageView*, juntando as características de cada um em um só lugar.

O *EditText* é uma extensão da *TextView* e permite aos usuários editar/entrar o texto através de uma entrada do teclado.

ListView é usada para exibir uma coleção de itens de forma linear em uma única coluna. Cada item individual pode ser selecionado para a exibição de mais detalhes ou realizar uma ação relacionada a esse item.

2.7. ESP *Touch*

ESP *touch* é um protocolo desenvolvido pela *espressif*, fabricante do microcontrolador ESP8266, o qual permite que um módulo se conecte à uma rede Wi-Fi sem saber previamente o *SSID* e o *password* da rede. O protocolo permite que um dispositivo *mobile* envie essas informações para o módulo *Wi-Fi* que ainda não está conectado à nenhuma rede. Em seguida módulo recebe essas informações através da técnica de *sniffing*, ou seja, quando um dispositivo que possui a tecnologia *Wi-Fi* procura por pacotes de dados que estejam trafegando no ar naquele momento.

2.7.1. Visão Geral da Tecnologia

O protocolo *ESP-TOUCH* da *espressif* implementa uma tecnologia de configuração inteligente para que usuários possam facilmente, utilizando seu *smartphone,* conectar dispositivos *IOT,* que utilizem o microcontrolador ESP8266, às suas redes *Wireless* domésticas através de uma configuração.

Como o dispositivo *ESP* não está conectado a rede *Wi-Fi* inicialmente, ou seja, o aplicativo não pode enviar os dados a ele diretamente.

Através do protocolo *ESP-TOUCH* um dispositivo *Wi-Fi,* como um *smartphone,* envia pacotes *UDP* para o ponto de acesso Wi-Fi e codifica o *SSID* e o *password* no campo '*length*' de uma sequência de pacotes UDP os quais o dispositivo ESP pode alcançar e decodificar.

Para o lado do dispositivo *ESP*, tanto o *sdk RTOS* (*Real Time Operacional System*) e o *non-os sdk* suportam o protocolo *ESP-TOUCH* e o fabricante fornece as

respectivas *APIs*. Para o lado do dispositivo *mobile* a fabricante *espressif* fornece um aplicativo de exemplo de código aberto para que possa ser usado em outras aplicações (FIGURA 18).

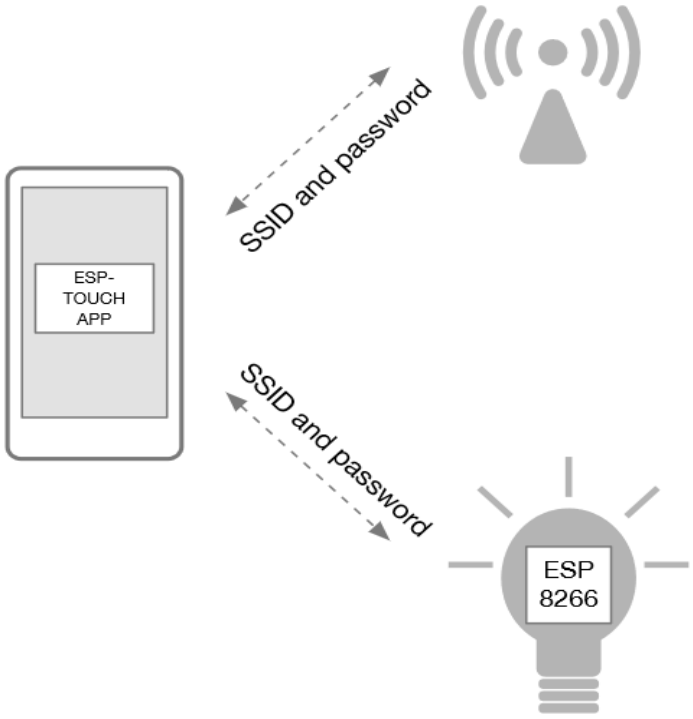

FIGURA 18. EXEMPLO APLICAÇÃO ESP- TOUCH.
FONTE: CORDEIRO (2016).

2.7.2. Processo de operação ESP-TOUCH

Os passos que precedem o processo de configuração do módulo são:

- Implementação da função *smart-config* no *firmware* do dispositivo *ESP*.
- Conectar o *smartphone* ao roteador.
- Instalar e rodar o aplicativo com o protocolo *ESP-TOUCH*.
- O aplicativo irá selecionar a rede a qual o *smartphone* esta conectado e o usuário apenas deve inserir o *password* e clicar no botão de configuração.

Terminado esses passos começará o processo de configuração do módulo. O *smartphone* irá enviar os pacotes por um determinado tempo e no lado do módulo o *ESP* irá escutar e tentar capturar tais pacotes por um determinado tempo. Quanto maior for a distância entre os dispositivos maior será o tempo consumido para que a configuração termine com sucesso.

Após isso o dispositivo *ESP* se conectar à rede e receber seu endereço *IP* ele irá retornar esse *IP* ao dispositivo *mobile*, que irá armazená-lo no sistema, atrelado ao módulo em questão.

2.7.3. APIs de desenvolvimento

Para desenvolver aplicações utilizando o protocolo *ESP-TOUCH* a fabricante fornece algumas *APIs* para desenvolvimento e utilização de tal protocolo.

A *smartconfig_start* configura o dispositivo e o conecta ao ponto de acesso *Wi-Fi*.

A instrução em C é:

- bool smartconfig_start(sc_callback_t cb, uint8 log) .

Os parâmetros são:

- sc_callback_t cb.
- Smart config callback, executada quando *smart config status* muda.
- uint8 log indica que o processo de conexão será mostrado via *UART (Receptor/Transmissor Universal Assíncrono)* ou não.

A função *'smartconfig_stop'* que deve ser chamada para parar o processo de *smart config* ou ao finalizar o processo, para que o buffer tomado pelo *smart config* seja liberado.

A instrução em C é:

- bool smartconfig_stop(void).

2.7.4. ESP8266 ESP-14

Para o desenvolvimento de um projeto sem a necessidade de um microcontrolador adicional, utilizando a linha ESP8266, é possível fazer o uso do módulo ESP-14, este apresenta algumas vantagens se comparado ao módulo básico.

O módulo Wi-Fi ESP-14 é composto pelo chip ESP8266 e é um módulo compacto, o que viabiliza a aplicação neste projeto. É ideal para projetos definitivos, pois contém 9 portas de *I/O* com funções de *PWM*, *I2C*, *SPI*, entre outras.

Com este módulo é possível ter diversas aplicações, como por exemplo:

- Tomadas inteligentes, automação residencial, monitoramento remoto, segurança doméstica, comercial e industrial;

- Rede de sensores, controle industrial sem fio, monitores de bebês e crianças, eletrônica vestível, dispositivos para localização via Wi-Fi;
- Tags de identificação para segurança, câmeras IP, robótica.

Especificações:

- Módulo Wireless ESP-14;
- Wireless padrão 802.11 b/g/n;
- Antena cerâmica e conector U-FL;
- Modos de operação: STA/AP/STA+AP;
- Protocolo TCP/IP integrado;
- Tensão de operação: 3,3V;
- Distância entre pinos: 2mm;
- Dimensões: 21,2 x 16 x 3,2 mm.

2.8. PROTOCOLO *MQTT*

O protocolo *MQTT* (*Message Queue Telemetry Transport*) é um protocolo de comunicação internet do tipo *M2M* (*machine-to-machine*), ou seja, um protocolo para comunicação de máquinas.

MQTT tem se tornado um padrão para aplicações *IOT* uma vez que é um protocolo leve, com possibilidade de rodar em dispositivos com baixa capacidade computacional e operar em ambientes de rede instáveis e com alta latência.

Este protocolo, ao contrário do protocolo *HTTP*, rompe com o paradigma cliente/servidor e utiliza um paradigma conhecido como p*ublish/subscriber*. Neste paradigma um dispositivo pode tanto operar como *publish,* como *subscriber* ou como os dois ao mesmo tempo. Quando atua como *publish* para um determinado tópico, significa que ele irá atualizar a informação mostrada nesse tópico, já dispositivos que atuam como um *subscriber,* estarão assinando para obter as informações de um determinado tópico.

Mensagens nesse protocolo são organizadas em tópicos e uma mensagem pode ser enviada para um determinado dispositivo ou para todos os dispositivos pertencentes àquele tópico. Como por exemplo, se desejar mandar uma mensagem para todas as lâmpadas na casa de Fulano terá um caminho de mensagem do tipo "Fulano/casa/lâmpada", porém se desejar mandar uma mensagem apenas para

a lâmpada do quarto de Fulano é possível mandar um caminho dessa forma "Fulano/casa/lampada/LampadaQuarto".

O *MQTT* é um protocolo de mensagens assíncrono, isso significa que é possível enviar e receber mensagens sem que o dispositivo recebedor esteja conectado à internet no mesmo momento em que a mensagem é enviada pelo dispositivo transmissor. Isso é importante para a indústria *IOT* já que em muitas aplicações os dispositivos não podem ficar 100% do tempo conectados por problemas de baixa qualidade na conexão com a rede, ou necessidade de economizar energia.

Através de um servidor *MQTT* centralizado e com *IP* fixo, é possível o fluxo de dados entre diferentes dispositivos que atuam como clientes, como dispositivos *IOT* e dispositivos *mobile*. Sem esse intermediador é impossível a troca de mensagens entre esses dispositivos já que eles possuem *IP* dinâmico e não podem ser acessados diretamente através de uma rede externa.

Para dispositivos de Internet das Coisas (*IOT*), a conexão com a Internet é um requisito. A conexão com a Internet permite que os dispositivos trabalhem entre si e com serviços de *back-end*. O protocolo de rede subjacente da Internet é o *TCP/IP*. Desenvolvido com base na pilha *TCP/IP*, o *MQTT* tornou-se o padrão para comunicações de *IOT*.

O MQTT foi inventado e desenvolvido inicialmente pela IBM no final dos anos 90. Sua aplicação original era vincular sensores em *pipelines* de petróleo a satélites. Como seu nome sugere, ele é um protocolo de mensagem com suporte para a comunicação assíncrona entre as partes. Um protocolo de sistema de mensagens assíncrono desacopla o emissor e o receptor da mensagem tanto no espaço quanto no tempo e, portanto, é escalável em ambientes de rede que não são confiáveis. Apesar de seu nome, ele não tem nada a ver com filas de mensagens, na verdade, ele usa um modelo de publicação e assinatura. No final de 2014, ele se tornou oficialmente um padrão aberto OASIS, com suporte nas linguagens de programação populares, usando diversas implementações de software livre.

O *MQTT* é um protocolo de rede leve e flexível que oferece o equilíbrio ideal para os desenvolvedores de *IOT*:

- O protocolo leve permite a implementação em hardware de dispositivo altamente restringido e em redes de largura da banda limitada e de alta latência.

- Sua flexibilidade possibilita o suporte a diversos cenários de aplicativo para dispositivos e serviços de *IOT*.

Para entender por que o *MQTT* é tão adequado para desenvolvedores de *IoT*, vamos analisar por que outros protocolos de rede populares falharam em *IoT*.

A maioria dos desenvolvedores já se acostumou aos serviços da *Web HTTP*. Então, por que não conectar os dispositivos de IoT aos serviços da web? O dispositivo poderia enviar seus dados como uma solicitação de *HTTP* e receber atualizações do sistema como uma resposta de *HTTP*. Esse padrão de solicitação e resposta tem algumas limitações graves:

- O *HTTP* é um protocolo síncrono. O cliente espera que o servidor responda. Os navegadores da web têm esse requisito, mas o custo é a baixa escalabilidade. No mundo da *IoT*, a comunicação síncrona tem sido um problema devido ao grande número de dispositivos e à rede, muitas vezes não confiável e de alta latência. Um protocolo de mensagem assíncrono é muito mais adequado para aplicativos de *IoT*. Os sensores podem enviar leituras e permitir que a rede descubra o caminho e a sincronização ideais para entregar aos dispositivos e serviços de destino.
- *HTTP* é unidirecional. O cliente precisa iniciar a conexão. Em um aplicativo de *IoT*, os dispositivos e sensores geralmente são clientes, o que significa que eles não podem receber comandos da rede passivamente.
- *HTTP* é um protocolo de um para um. O cliente faz uma solicitação e o servidor responde. É difícil e caro transmitir uma mensagem a todos os dispositivos na rede, o que é um caso de uso comum em aplicativos de *IoT*.
- *HTTP* é um protocolo pesado com muitos cabeçalhos e regras. Ele não é adequado para redes restringidas.

Pelos motivos citados acima, a maioria dos sistemas escaláveis de alto desempenho usam um barramento do sistema de mensagens assíncrono, em vez de serviços da web, para trocas de dados internas. Na verdade, o protocolo de sistema de mensagens mais popular que é usado em sistemas de *middleware* corporativos é chamado *AMQP* (*Advanced Message Queuing Protocol*). No entanto,

no ambiente de alto desempenho, a capacidade de computação e a latência da rede geralmente não são uma preocupação. O *AMQP* foi criado para assegurar a confiabilidade e a interoperabilidade em aplicativos corporativos. Ele possui um rico conjunto de recursos, mas não é adequado para aplicativos de *IoT* com restrição de recursos.

Além do *AMQP*, existem outros protocolos populares de sistema de mensagens. Por exemplo, o *XMPP (Extensible Messaging and Presence Protocol)* é um protocolo de mensagem instantânea *(IM)* ponto a ponto. Ele é pesado em recursos com suporte para casos de uso de IM, como presença e anexos de mídia. Em comparação com o *MQTT*, ele requer muito mais recursos no dispositivo e na rede.

Um importante recurso do protocolo *MQTT* é o modelo de publicação e assinatura. Como em todos os protocolos de sistema de mensagens, ele desacopla o publicador e o consumidor de dados.

O protocolo *MQTT* define dois tipos de entidades na rede: um *message broker* e inúmeros clientes. O *broker* é um servidor que recebe todas as mensagens dos clientes e, em seguida, roteia essas mensagens para os clientes de destino relevantes. Um cliente é qualquer coisa que possa interagir com o *broker* e receber mensagens. Um cliente pode ser um sensor de *IoT* em campo ou um aplicativo em um *data center* que processa dados de *IoT*.

O cliente conecta-se ao *broker*. Ele pode assinar qualquer "tópico" de mensagem no broker. Essa conexão pode ser uma conexão *TCP/IP* simples ou uma conexão *TLS* criptografada para mensagens sensíveis.

O cliente publica as mensagens em um tópico, enviando a mensagem e o tópico ao *broker*.

Em seguida, o *broker* encaminha a mensagem a todos os clientes que assinam esse tópico.

Como as mensagens do *MQTT* são organizadas por tópicos, o desenvolvedor de aplicativos tem a flexibilidade de especificar que determinados clientes somente podem interagir com determinadas mensagens. Por exemplo, os sensores publicarão suas leituras no tópico "*sensor_data*" e assinarão o tópico "*config_change*". Os aplicativos de processamento de dados que salvam os dados do sensor em um banco de dados de *backend* assinarão o tópico "*sensor_data*". Um aplicativo de console administrativo poderia receber comandos do administrador do

sistema para ajustar as configurações dos sensores, como a sensibilidade e a frequência de amostragem, e publicar essas mudanças no tópico "config_change".

Ao mesmo tempo, o *MQTT* é leve. Ele tem um cabeçalho simples para especificar o tipo de mensagem, um tópico baseado em texto e, em seguida, uma carga útil binária arbitrária. O aplicativo pode usar qualquer formato de dados para a carga útil como *JSON, XML*, binário criptografado ou Base64, desde que os clientes de destino possam analisar a carga útil.

3. DESENVOLVIMENTO

3.1. APLICATIVO ANDROID

A interação do usuário com o sistema é feita por meio de um *software* desenvolvido especificamente para essa função em linguagem de programação Java, que opera em sistemas Android 4.0 ou superior.

Este *software* será implementado utilizando a plataforma Android Studio 2.0 (FIGURA 19), uma IDE muito utilizada para o desenvolvimento de aplicativos para sistemas Android.

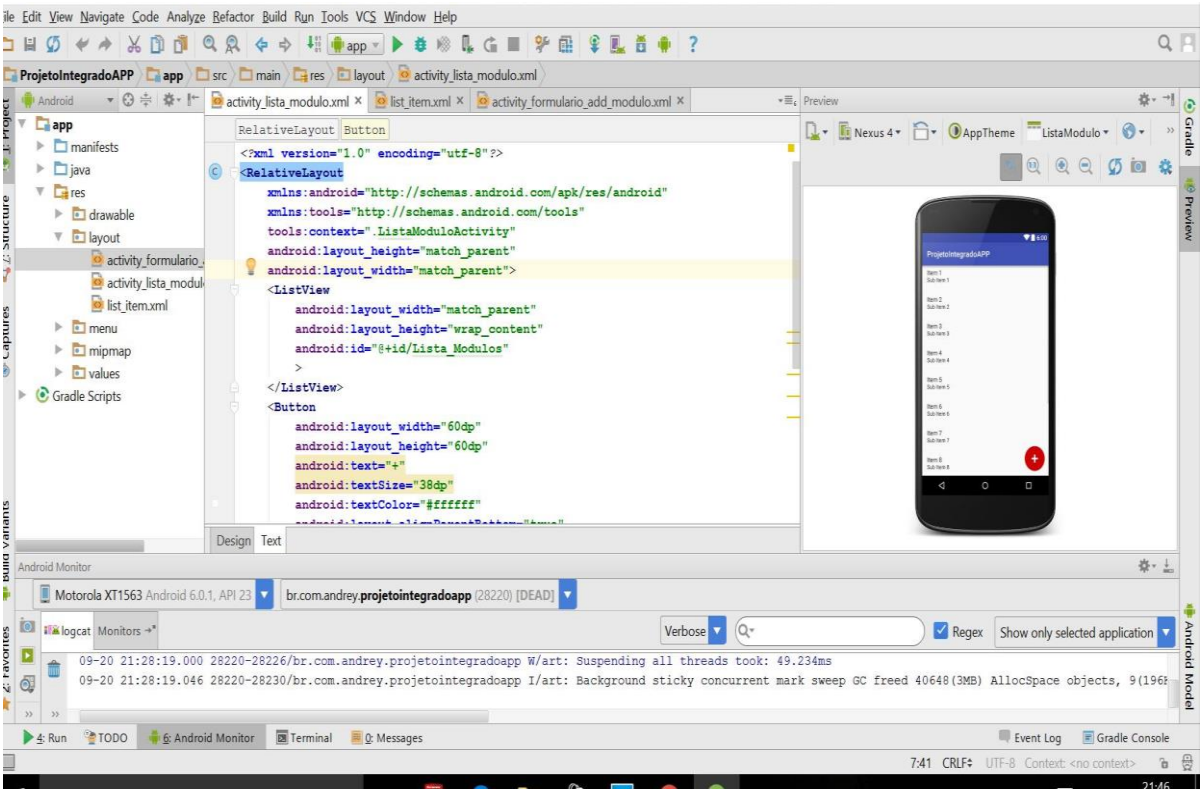

FIGURA 19. PRINTSCREN DO *SOFTWARE* ANDROID STUDIO
FONTE: AUTOR (2019).

3.1.1. Funcionamento do Aplicativo

O aplicativo é constituído por duas telas: lista dos módulos já adicionados e uma tela para adicionar novos módulos ao sistema.

A FIGURA 20 mostra o esquemático de desenvolvimento e funcionamento do aplicativo.

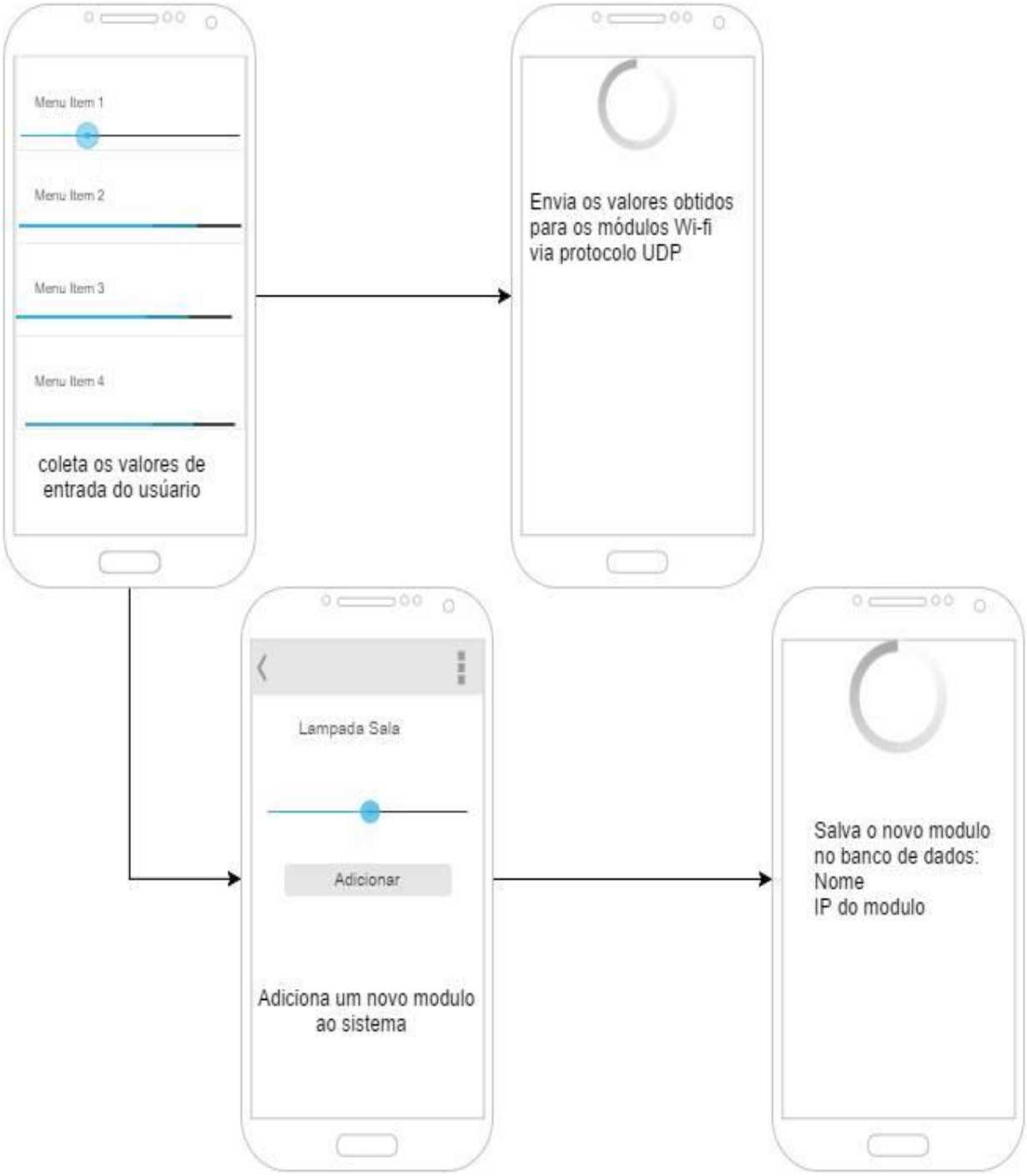

FIGURA 20. FUNCIONAMENTO DO APLICATIVO
FONTE: AUTOR (2018).

A tela inicial do aplicativo apresenta uma lista de todos os módulos já adicionados ao sistema e um botão com a opção de adicionar novos módulos.

Cada item da lista contém o nome do determinado módulo e uma barra de progresso de entrada de dados do sistema determinada pelo usuário. A barra de progresso fornece um valor de 0 a 100 baseado em sua posição. Este valor será responsável por controlar a intensidade luminosa de uma lâmpada ou a posição de um motor de passo em certo percurso, por exemplo.

No momento do arrasto da barra de progresso, o aplicativo abre uma *Thread* responsável por enviar o valor de progresso da barra para o módulo de automação, via protocolo *MQTT*.

O botão para inserir novos módulos na primeira tela, leva o usuário a uma segunda tela que contém um formulário para adicionar novos módulos de automação, ali o usuário insere o nome que deseja dar a esse módulo e o progresso da barra caso queira iniciar o dispositivo com algum valor inicial.

Ao adicionar o novo módulo ao sistema, o aplicativo guarda as novas informações em seu banco de dados interno para que a configuração não seja perdida após o desligamento do aplicativo.

3.1.2. Configurações gráficas

As configurações gráficas são configurações das *views* do Android, ou seja, são classes responsáveis por exibir itens na tela que podem interagir com o usuário ou não, como já visto no capítulo anterior. Algumas classes básicas já são disponibilizadas pela API do Android e suas bibliotecas, porém para atender as demandas específicas dentro do sistema é necessário criar novas *views*.

Nessa seção serão apresentadas as configurações de algumas *views* nativas do Android e algumas criadas, e o uso de bibliotecas externas customizadas para atender à esse projeto.

A aplicação básica deste sistema contém em sua tela inicial uma lista de módulos e um botão para adicionar novos módulos, a seguir será apresentado como é feito a configuração da lista de *views* e do botão.

Para exibir listas e botões no Android, deve-se criar *views* em um *XML* dentro do Android Studio (FIGURAS 21 e 22). No *XML* responsável pela configuração da tela temos duas *views*, uma é a ListView responsável por exibir uma lista na tela e a outra é um *Button* responsável por exibir um botão. No arquivo *XML* é configurada

toda a parte gráfica das *views*, os comportamentos dos itens serão configurados nas classes do Java.

Pela FIGURA 21 é possível observar que o *XML* está configurando o botão, bem como sua cor, tamanho, tamanho do texto, texto e posição na tela, e também uma lista de itens simples. A lista do nosso aplicativo contém outras *views* que descrevem as aparências dos itens, e está em outro arquivo *XML*.

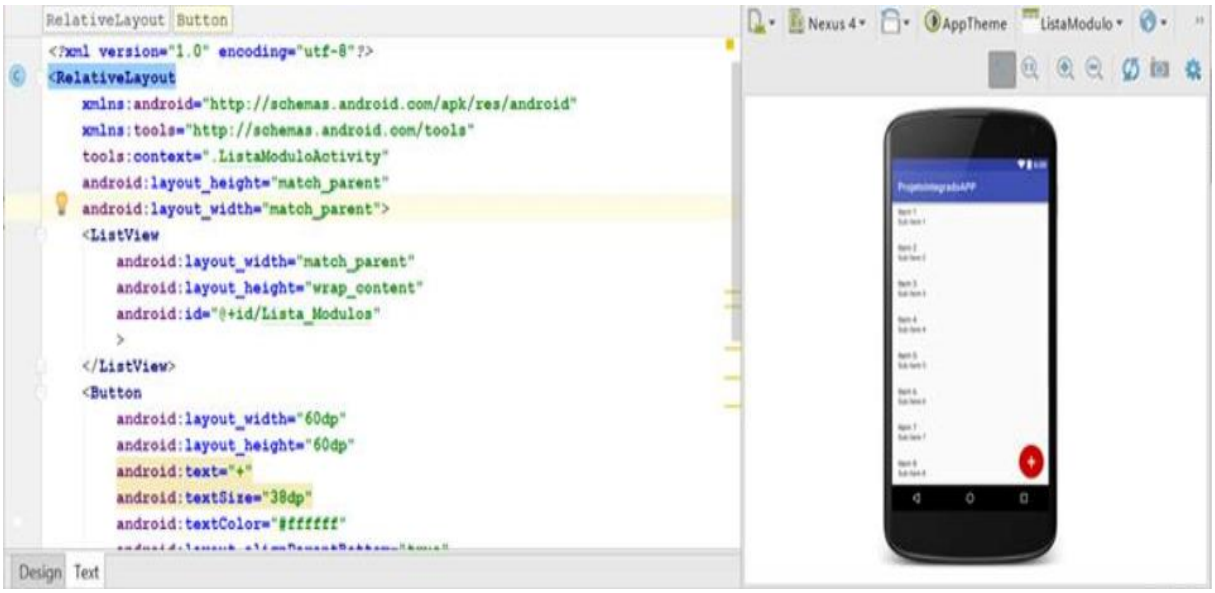

FIGURA 21. *CONFIGURAÇÃO XML*

FONTE: AUTOR (2019).

FIGURA 22. CONFIGURAÇÃO *XML*

FONTE: AUTOR (2019).

A FIGURA 22 apresenta a configuração da aparência do item da lista, onde basicamente tem-se duas *TextViews* para indicar o começo e o fim da barra de progresso, a *SeekBar*, o tamanho, texto, a cor e a posição de cada uma dessas *views* que são configuradas em *tags*.

Além das views padrão da API do Android vistas anteriormente, também são utilizadas nesse projeto *views* criadas externamente à API Android para atender problemas específicos. Um caso de aplicação de uma biblioteca de *views* externa neste projeto é para melhorar a usabilidade e design do aplicativo para as atividades relacionadas ao módulo de iluminação com o controle de um led WRGB, por exemplo. A biblioteca utilizada é chamada de ColorPicker, ou seja, é uma palheta de cores e algumas views auxiliares que facilitam a manipulação e a escolha das cores por parte do usuário para controlar o módulo do led WRGB.

Esta biblioteca de views possui um total de 9 classes. Dentre classes para grupo de *views*, para a *view* especifica e para o tráfego de informações entre as *views* e o sistema. Essas classes serão apresentadas a seguir, descrevendo seu funcionamento e sua utilidade para o sistema.

A classe ObservableColor é responsável pelo tráfego de informações e por guardar o estado de um determinado objeto do tipo ColorObserver, ou seja, uma interface da biblioteca implementada aos objetos que terão uma cor para ser observada.

A classe ObservableColor armazena os dados referentes a cor de um objeto no formato HSV e um segundo parâmetro Alpha. Esse formato HSV é um composto por 3 parâmetros: o H que remete a hue e é a matriz de cor, S de saturação e V de Value e faz referência a intensidade da luz. O segundo parâmetro Alpha a principio é responsável pela transparência da cor, no caso deste projeto o Alpha é utilizado para controlar a luz branca do led WRGB.

Como pode ser visto na FIGURA 23, a classe ObservableColor armazena o dado de cor em um array chamado HSV e uma variável independente chamada de Alpha, além de alguns métodos que auxiliam na programação para que os dados possam ser tratados de uma maneira diferente. Ao invés de utilizar o modelo HSV é possível utilizar o modelo tradicional de cores em valores que variam de 0 a 255 através do método getColor ou utilizar informações especificas do HSV como em outros métodos da classe que simplesmente retornam o valor correspondente do array.

```
public class ObservableColor {

    private final float[] hsv = {0, 0, 0};
    private int alpha;
    private final List<ColorObserver> observers = new ArrayList<~>();

    public ObservableColor(int color) {
        Color.colorToHSV(color, hsv);
        alpha = Color.alpha(color);
    }

    public void getHsv(float hsvOut[]) {
        hsvOut[0] = hsv[0];
        hsvOut[1] = hsv[1];
        hsvOut[2] = hsv[2];
    }

    public int getColor() { return Color.HSVToColor(alpha, hsv); }

    public float getHue() { return hsv[0]; }

    public float getSat() { return hsv[1]; }

    public float getValue() { return hsv[2]; }

    public int getAlpha() { return alpha; }

    public float getLightness() { return getLightnessWithValue(hsv[2]); }
```

FIGURA 23. CLASSE OBSERVABLECOLOR
FONTE: AUTOR (2019).

Para que um objeto possa ter sua cor observada, ou seja, ter a classe ObservableColor associada a ele é necessário que ele implemente a interface chamada ColorObserver a qual obriga a implementação do método UpdateColor na classe correspondente ao objeto (FIGURA 24).

```
public interface ColorObserver {
    void updateColor(ObservableColor observableColor);
}
```

FIGURA 24. MÉTODO UPDATECOLOR NA CLASSE OBSERVABLECOLOR
FONTE: AUTOR (2019).

Outra classe operacional da biblioteca ColorPicker é a classe Resources. Essa é uma classe responsável por armazenar parâmetros de configurações e métodos de auxílio para fazer o desenho das views (FIGURA 25).

```
class Resources {

    private static final float LINE_WIDTH_DIP = 1.5f;
    private static final float POINTER_RADIUS_DIP = 7;
    private static final int VIEW_OUTLINE_COLOR = 0xff808080;

    public static Paint makeLinePaint(Context context) {
        Paint paint = new Paint();
        paint.setColor(VIEW_OUTLINE_COLOR);
        paint.setStrokeWidth(dipToPixels(context, LINE_WIDTH_DIP));
        paint.setStyle(Paint.Style.STROKE);
        paint.setAntiAlias(true);
        return paint;
    }

    public static Paint makeCheckerPaint(Context context) {
        Paint paint = new Paint();
        final Bitmap checkerBmp = BitmapFactory.decodeResource(context.getResources(), R.drawable.background);
        paint.setShader(new BitmapShader(checkerBmp, Shader.TileMode.REPEAT, Shader.TileMode.REPEAT));
        paint.setStrokeWidth(dipToPixels(context, LINE_WIDTH_DIP));
        paint.setStyle(Paint.Style.FILL);
        paint.setAntiAlias(true);
        return paint;
    }

    public static Path makePointerPath(Context context) {
        Path pointerPath = new Path();
        final float radiusPx = dipToPixels(context, POINTER_RADIUS_DIP);
        pointerPath.addCircle(0, 0, radiusPx, Path.Direction.CW);
        return pointerPath;
```

FIGURA 25. CLASSE RESOURCES

FONTE: AUTOR (2019).

As próximas classes são referentes a implementação das *views* em si, três delas podem ser utilizadas no projeto para representar as funcionalidades do módulo de iluminação com o led WRGB. Algumas views são um *viewgroup*, ou seja, são apenas a base para a construção de outras views.

A classe *AlphaView*, nesse projeto será utilizada para controlar a cor branca do LED WRGB, essa classe como dito anteriormente implementa a interface ColorObserver, o que significa que os objetos ligados a essa classe estarão observando e guardando o registro de uma cor específica. Esta classe também apresenta métodos responsáveis por gerar um desenho da *view*, que no caso é um desenho retangular com um degrade de cores do branco total até a cor pura escolhida pelo usuário. Um pedaço de código importante, chamado de Listener, teve de ser adicionado à essa classe da biblioteca, ele é responsável por perceber o momento que o usuário toca na *view* e então executar uma tarefa especifica. Para

tal é necessário adicionar à classe da *view* uma interface privada que possui o método que deve ser executado no momento em que o usuário toca na view que está disposta na tela do dispositivo *mobile* (FIGURA 26).

```
public class AlphaView extends SliderViewBase implements ColorObserver {
    //////////////Listener
    public interface OnAlphaChangeListener{
        void onAlphaChanged(ObservableColor observableColor);
    }
    AlphaView.OnAlphaChangeListener mOnAlphaChangeListener;
    public void setOnAlphaChangeListener(AlphaView.OnAlphaChangeListener l){
        mOnAlphaChangeListener = l;
    }
    private ObservableColor observableColor = new ObservableColor(0);

    public AlphaView(Context context) { this(context, null); }

    public AlphaView(Context context, AttributeSet attrs) { super(context, attrs); }

    public void observeColor(ObservableColor observableColor) {
        this.observableColor = observableColor;
        observableColor.addObserver(this);
    }

    @Override
    public void updateColor(ObservableColor observableColor) {
        setPos((float)observableColor.getAlpha()/0xff);
        updateBitmap();
        invalidate();
    }

    @Override
    protected void notifyListener(float currentPos) {
        observableColor.updateAlpha((int)(currentPos * 0xff), this);
```

FIGURA 26. CLASSE ALPHAVIEW
FONTE: AUTOR (2019).

Outra *view* bastante parecida com a *AlphaView* é a *ValueView*, esta *view* tem funcionamento e aparência bastante semelhante a *AlphaView*, ambas são retângulos com degrade de cores. Porém a *ValueView* possui uma função um pouco diferente dentro deste sistema, que é a função de determinar a intensidade de luz a qual o led deve estar operando. O mesmo procedimento com relação ao Listener teve de ser feito com esta view também (FIGURA 27).

```java
public class ValueView extends SliderViewBase implements ColorObserver {
    public interface OnValueChangeListener{
        void onValueChanged(ObservableColor observableColor);
    }
    OnValueChangeListener mOnValueChangeListener;
    public void setOnValueChangeListener(OnValueChangeListener l) { mOnValueChangeListener = l; }
    ////////////////////////
    private ObservableColor observableColor = new ObservableColor(0);

    public ValueView(Context context) { this(context, null); }

    public ValueView(Context context, AttributeSet attrs) { super(context, attrs); }

    public void observeColor(ObservableColor observableColor) {
        this.observableColor = observableColor;
        observableColor.addObserver(this);
    }

    @Override
    public void updateColor(ObservableColor observableColor) {
        this.observableColor = observableColor;
        setPos(this.observableColor.getValue());
        updateBitmap();
        invalidate();
    }

    @Override
    protected void notifyListener(float currentPos) {
        observableColor.updateValue(currentPos, this);
```

FIGURA 27. CLASSE VALUEVIEW

FONTE: AUTOR (2019).

A última *view* utilizada no projeto é a *view* responsável pelo desenho da palheta de cores em si. Esta possui um formato de quarto de circulo o que facilita a seleção da cor do meio, no caso a cor branca, e oferece um degrade de cores passando por todas as cores possíveis (FIGURA 28). Essa *view* é chamada de *HueSatView* e também implementa o *Listener* para que o aplicativo possa executar trechos de código no momento em que o usuário está tocando essa *view* especifica.

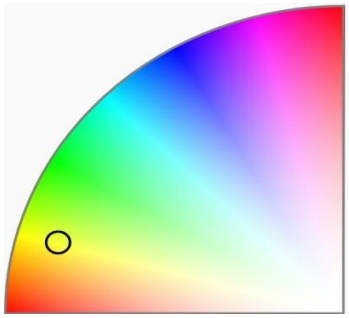

FIGURA 28. VIEW HUESATVIEW

FONTE: AUTOR (2019).

Para que essas *views* sejam dispostas na tela para o usuário é necessário configurar suas posições e tamanhos no arquivo XML correspondente a *activity* que implementará essas *views*. Na FIGURA 29 é possível observar a configuração dessas três *views* em um arquivo XML.

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Controle luz branca"/>

    <br.com.andrey.projetointegradoapp.ColorPicker.AlphaView
        android:layout_width="200dp"
        android:layout_height="50dp"
        android:id="@+id/alpha_view"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="choose a color"/>

    <br.com.andrey.projetointegradoapp.ColorPicker.HueSatView
        android:layout_width="200dp"
        android:layout_height="150dp"
        android:id="@+id/hue_sat"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Controle intensidade luz"/>

    <br.com.andrey.projetointegradoapp.ColorPicker.ValueView
        android:layout_width="300dp"
        android:layout_height="50dp"
        android:id="@+id/value_view"/>
</LinearLayout>
```

FIGURA 29. CONFIGURAÇÃO DAS VIEWS NO XML
FONTE: AUTOR (2018).

3.1.3. Código em Linguagem de Programação Java

Esse tópico tem o objetivo de listar os principais trechos do código Java bem como uma breve explicação do funcionamento de cada comando e função dentro das classes do sistema.

O comportamento do sistema do aplicativo é todo descrito em linguagem de programação Java, bem como o comportamento dos itens que aparecem na tela são configurados em uma classe do Java.

O paradigma que governa linguagens de programação como o JAVA é conhecido como orientação a objeto, que nada mais é, que uma forma de escrever código de forma a modelar o sistema em classes e objetos, tornando o código mais reutilizável e de fácil manutenção.

No sistema que aqui se trata, a classe chave para modelar nosso sistema é a classe Modulo, é esta classe que permitirá abstrair a essência do que é um módulo de automação para o sistema, essa classe representa o que é comum a todos os diferentes módulos do nosso sistema e guardará informações relativas a esses módulos como nome, endereço IP, por exemplo. A classe Modulo é uma classe abstrata e serve de base para outras classes especificas para cada módulo, que são todas classes filha da classe Modulo, ou seja, elas herdam da classe Modulo todos os seus comportamentos e atributos, além de implementarem o seus respectivos comportamentos específicos.

Para o comportamento da tela inicial do aplicativo desenvolvido foram feitas algumas configurações para as ações no momento do clique no botão e para o clique na lista (FIGURA 30).

Até o momento o sistema abriga três diferentes módulos, que são representados pelas classes ModuloDimmer, ModuloLedRGB, e ModuloSwitch.

Essas são as classes que são usadas no sistema, elas implementam todas as características da classe mãe a classe Modulo, e os seus comportamentos específicos como a classe ModuloDimmer apenas tem a necessidade de guardar o valor do PWM que está aplicado ao dimmer para o controle de uma lâmpada.

A classe ModuloLedRGB precisa guardar os valores que será aplicado ao PWM de cada um dos quatro leds, azul, branco, verde e vermelho, já a classe ModuloSwitch precisa apenas guardar a variável booleana que informa se a chave esta aberta ou fechada.

Todas as classes usadas para descrever o comportamento direto da tela são extendidas de uma classe mãe chamada *AppCompatActivity*, isso significa que essa classe representa uma *activity* da tela do aplicativo. Esta implementa alguns métodos importantes como *OnCreate,* que utiliza o sistema operacional para plotar os dados na tela para o usuário, bem como outros métodos opcionais como o *OnCreateContextMenu*, que cria um menu de contexto para dar certas opções ao usuário, como por exemplo, editar ou excluir um item de uma lista.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_lista_modulo);

    listaModulo = (ListView) findViewById(R.id.Lista_Modulos);

    listaModulo.setOnItemClickListener((parent, view, position, id) -> {
            modulo =  (Modulo) listaModulo.getItemAtPosition(position);
            Toast.makeText(ListaModuloActivity.this, modulo.getProgress() + "%", Toast.LENGTH_SHORT).show();

    });

    Button addModulo = (Button) findViewById(R.id.Lista_Botao_adiciona);
    addModulo.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            Intent novoModulo = new Intent(ListaModuloActivity.this, FormularioAddModuloActivity.class);
            startActivity(novoModulo);
        }
    });
    registerForContextMenu(listaModulo);
}
```

FIGURA 30. CONFIGURAÇÃO TELA INICIAL

FONTE: AUTOR (2019)

Dentro do método *OnCreate* é inflado o *XML* responsável pelo *layout* da tela *ListaModulos*, então é criada uma variável para cada *view* que será manipulada. A partir dessa variável, ou instância da *view*, é configurado o comportamento para um clique, que no caso do botão é criar uma *intent* e iniciar uma nova *activity*, enquanto que no caso do clique na lista é mostrado o valor da barra de progresso em uma caixa de dialogo, chamada no Android de *Toast*.

A *ListView* é uma lista simples, a princípio vazia, e para popular essa lista é necessário criar um *Adapter* (classe no Android que organiza os itens da lista que devem aparecer em determinado momento na tela).

Na classe chamada de *ModuloAdapter* será configurado o comportamento das views que estão dentro do item da lista.

No momento em que a *view SeekBar* é arrastada e seu valor muda, o método *onProgressChanges* é executado pelo sistema. Dentro desse método é verificado se a barra foi modificada pelo usuário, em caso afirmativo é instanciada uma *Thread* que irá executar a classe *UDP* em paralelo à execução do resto do programa. No construtor dessa classe é enviado o valor da barra de

progresso e o *IP* para o qual o programa deve enviar os dados, então a Thread é iniciada.

O comportamento mais importante é o que será executado quando houver alguma mudança no arrasto da barra de progresso (FIGURA 31).

```
holder.campoBarra.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser) {
        if(fromUser) {
            modulo.setProgress(progress);
            Thread t = new Thread(new UDP(progress, modulo.getModuleIpAdress()));
            t.start();
        }
    }
```

FIGURA 31. CÓDIGO BARRA DE PROGRESSO
FONTE: AUTOR (2019)

A classe *UDP* é um *socket UDP/IP* que é responsável por fazer os envios dos pacotes *UDP* ao *IP* pré-determinado. Como essa classe vai rodar em uma *Thread*, ou seja, em paralelo ao resto do sistema, a classe é uma extensão da classe da linguagem Java, *Runnable* (FIGURA 32).

```
public class UDP implements Runnable {
    private final int progress;
    private final String IPadress;

    public UDP(int progress, String moduleIpAdress) {
        this.progress = progress;
        this.IPadress = moduleIpAdress;
    }

    @Override
    public void run() {
            try {
                String messageStr = ""+progress;
                int server_port = 9876;
                DatagramSocket s = new DatagramSocket();
                InetAddress local = InetAddress.getByName(this.IPadress);
                int msg_length = messageStr.length();
                byte[] message = messageStr.getBytes();
                DatagramPacket p = new DatagramPacket(message, msg_length, local,
                        server_port);
                s.send(p);
            }
```

FIGURA 32. CONFIGURAÇÃO *UDP/RUNNABLE*
FONTE: AUTOR (2019)

Como o envio de informações pela internet pode ser lento e ocorrer problemas inesperados, é necessário que essa parte do programa rode em uma *Thread* e sua execução fique dentro de um bloco *try* que irá tentar executar esse trecho do código, e caso tenha algum problema, como falta de internet, ele irá tratar essa exceção ao invés de fechar o programa abruptamente. A implementação desse *socket UDP* é relativamente simples, só é necessário informar o que será enviado em formato *string*, a porta na qual será estabelecida a conexão e o *IP* do recebedor da mensagem.

No sistema operacional Android algumas chamadas ao sistemas pedem permissão do usuário para serem feitas, o uso da internet é uma permissão que deve ser concedida pelo usuário no momento da instalação do aplicativo. Portanto, se faz necessário informar ao usuário que esse aplicativo faz chamadas à internet e para que isso ocorra é necessário adicionar uma permissão no arquivo de configuração geral e informações importantes do aplicativo, o arquivo Manifest.xml (FIGURA 33).

Para adicionar a permissão de uso de internet é necessário adicionar a *tag uses-permission*, para o uso de internet. Com isso, o aplicativo já está pronto para enviar pacotes *UDP* contendo o valor da barra de progressos.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.com.andrey.projetointegradoapp">

    <uses-permission android:name="android.permission.INTERNET"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="ProjetoIntegradoAPP"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".ListaModuloActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".FormularioAddModuloActivity"></activity>
    </application>

</manifest>
```

FIGURA 33. CONFIGURAÇÃO PERMISSÃO DE ACESSO A INTERNET

FONTE: AUTOR (2019)

Para adicionar a permissão de uso de internet é necessário adicionar a *tag uses-permission*, para o uso de internet. Com isso, o aplicativo já está pronto para enviar pacotes *UDP* contendo o valor da barra de progressos.

No Android, quando uma aplicação cria uma nova variável ou reserva algum espaço na memoria, se faz isso em uma memoria temporária que é descartada após o fechamento da aplicação. Para garantir que os módulos adicionados não desapareçam da aplicação após o usuário fechar o aplicativo deve-se garantir que certos dados sejam gravados em uma memoria permanente. Para isso é necessário à criação de um banco de dados interno (FIGURA 34).

```
public class ModuloDAO extends SQLiteOpenHelper {
    public ModuloDAO(Context context) { super(context, "modulos", null, 3); }

    @Override
        public void onCreate(SQLiteDatabase db) {
            String sql = "create table modulos(id integer primary key, nome text not null, progress double)";
            db.execSQL(sql);
        }
```

FIGURA 34. CÓDIGO BANCO DE DADOS INTERNO
FONTE: AUTOR (2019).

O banco de dados é configurado em uma nova classe do Android chamada de uma classe filha *SQLiteOpenHelper*, que é usada para criar um banco de dados baseado na linguagem *SQL*. No método *OnCreate* é executada a *query* que cria a tabela desejada, onde a *query* cria uma nova tabela chamada "módulos", sendo a primeira coluna destinada ao *ID* do módulo, a segunda coluna ao nome que o usuário deseja dar para o determinado módulo e a posição da barra de progresso.

3.2. DISPOSITIVOS

Os dispositivos trabalham em conjunto com o processamento, recebendo informações do usuário e dos sensores, tomando alguma decisão e a implementando na saída através dos atuadores.

A princípio, o projeto irá conter três dispositivos que realizarão tarefas genéricas de automação, como controle de iluminação, controle de relés e

controle de motor de passo. Porém a ideia do projeto é que módulos possam ser criados conforme demanda e acoplados ao sistema, de maneira independente dos outros módulos.

Para fazer o controle de iluminação do ambiente, será implementado um circuito *dimmer* capaz de variar a potência de uma lâmpada, regulando a intensidade por meio de um sinal *PWM* na entrada. Para esse módulo também poderá ser adicionado um sensor para medir a corrente e tensão eficaz, assim o circuito poderá retornar ao usuário o consumo com essa lâmpada em especifico.

Outra implementação para automação, de forma genérica, é o uso de um relé controlado via *software*. Esse dispositivo tem como função apenas ligar ou desligar um determinado eletrodoméstico, sem implementação de um controle de potência. Tanto neste dispositivo como no *dimmer* é possível implementar sensores de corrente e tensão a fim de monitorar a rede e detectar sobrecargas ou curto-circuitos, podendo atuar diretamente na falha, desligando o dispositivo.

A implementação de novos dispositivos é escalável, então pode ser adicionadas novas funcionalidades ao sistema usando como base a estrutura já existente. Um novo dispositivo apenas precisa receber um valor qualquer na entrada e modificar a saída, além de ser capaz de coletar informações do ambiente através de diversos tipos de sensores que podem ser utilizados.

3.3. SENSORES

Diversos sensores podem ser utilizados junto ao sistema para aumentar a eficiência dos processos e permitir a automatização de funções.

3.3.1. Sensor de gás

Para estimular o desenvolvimento de sistemas de segurança, também fará parte desse projeto um módulo capaz de realizar gases de um ambiente. Para isso, será utilizado o sensor MQ-2, capaz de detectar diversos gases combustíveis e fumaça, tais como: GLP, metano, propano, butano, hidrogênio, álcool, gás natural e outros inflamáveis.

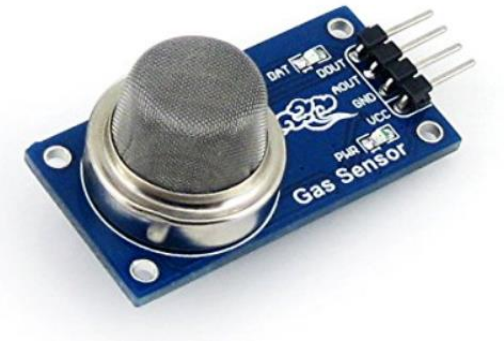

FIGURA 35. SENSOR DE GÁS
FONTE: AMAZON (2019).

3.4. ATUADORES

Os atuadores são as partes do sistema que executaram alguma função final, ou seja, são circuitos que fazem a abertura ou fechamento da janela, transferem um sinal PWM para uma lâmpada ou acionam um relé.

3.4.1. Driver Dimmer

O *driver Dimmer* (FIGURA 36) é um bloco de circuito responsável por receber um sinal *PWM* e acionar uma lâmpada, controlando a potência de saída de acordo com a abertura do *PWM*.

Esse *driver* pode ser implementado de diversas formas, vários circuitos diferentes oferecem soluções parecidas para esse problema. O circuito escolhido utiliza um opto acoplador para isolar o sinal *PWM*, que vem da saída do microcontrolador, do sinal da rede que alimenta a lâmpada. Um *mosfet BUZ41A* é encarregado de fazer o chaveamento de potência da lâmpada.

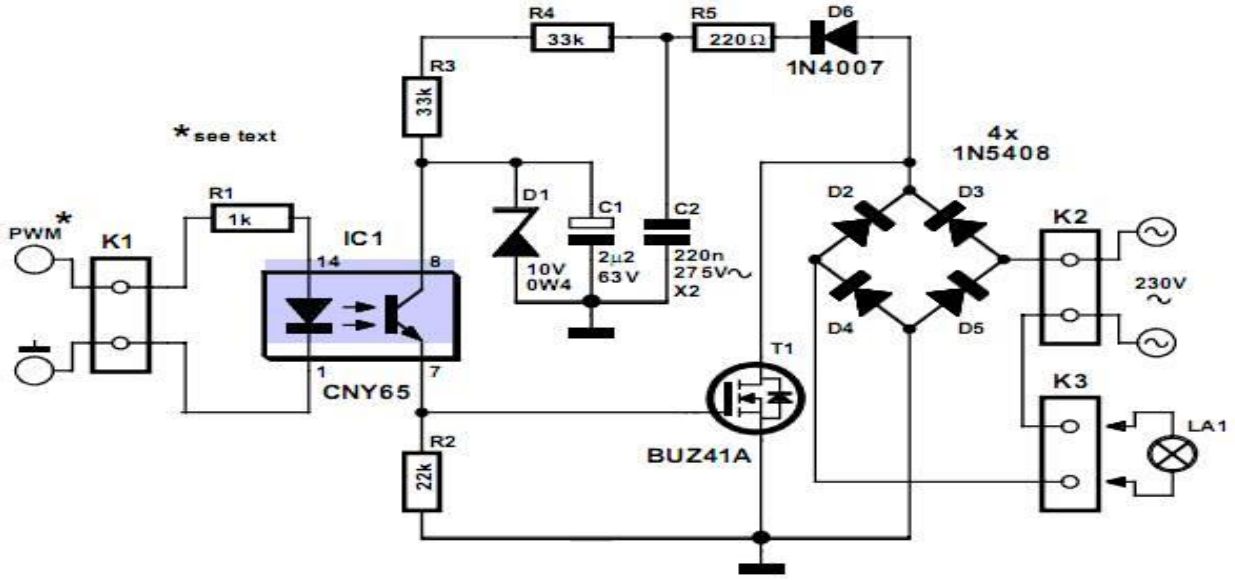

FIGURA 36. CIRCUITO DRIVER DIMMER BUZ41A
FONTE: NOVA ELETRÔNICA (2015).

O diodo D5, o resistor R5, e o capacitor C2 da FIGURA 36, formam um retificador com filtro para polarizar tanto o *mosfet* como o transistor do opto acoplador. O diodo zener D1 regula a tensão no ponto sobre o transistor do opto acoplador em 10 volts. Por fim, o resistor R2, um resistor de resistência alta, 22 kΩ serve para conter correntes de pico que podem surgir devido ao rápido chaveamento e indutâncias parasitas.

3.4.2. Driver Relé

O driver relé (FIGURA 37) é o responsável por acionar uma chave eletro mecânica, conhecida como relé. O acionamento desse dispositivo é bem mais simples que os anteriores.

O circuito representa o acionamento de um relé. É necessário apenas um *transistor* para amplificar a corrente de entrada, já que o microcontrolador não é capaz de fornecer corrente suficiente para acionar o relé. O diodo em paralelo no circuito serve para que quando o indutor de acionamento estiver carregado, o indutor se descarregue por ele próprio afim de não gerar uma corrente reversa sobre o transistor, que danificar o transistor.

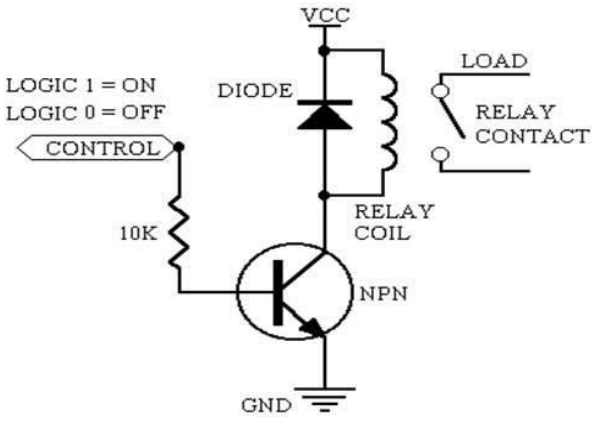

FIGURA 37. CIRCUITO DE ACINAMENTO DE UM RELÉ.
FONTE: METALTEX (2016).

3.4.3. Driver LED de Potência *RGB*

O módulo realiza o acionamento de um *LED* de potência *RGB*, através de um sinal *PWM*.

O *driver* de acionamento para o *LED* de potencia é um circuito que transforma o sinal *PWM* do microcontrolador, de baixa potência, em um sinal *PWM* de potência suficiente para acionar o dispositivo.

Esse circuito implementa três fases para o aumento da tensão e corrente, cada uma, ao final, alimenta um *LED* com o sinal *PWM* amplificado, podendo ser um *LED* vermelho, verde ou azul. A combinação desses três *LEDs* acionados com diferentes configurações de largura de pulso *PWM*, é capaz de produzir uma escala de 16 milhões de cores, já que cada cor varia de acordo com um valor de um *byte*, ou seja, varia de um valor de 0 a 255 em ambos os três *LEDs*.

A FIGURA 38 apresenta o circuito desenvolvido para o acionamento do *PWM* para os dispositivo deste projeto.

A FIGURA representa o esquemático de um *driver PWM* para um *LED*. Como a ideia é utilizar três *LEDs* que devem receber os sinais *PWM* simultâneos e independentes, é preciso replicar o mesmo circuito de polarização dos transistores Q1 e Q2. Também se deve replicar o acionamento do transistor de potência M1, para quantos sinais forem necessários, nesse caso, três.

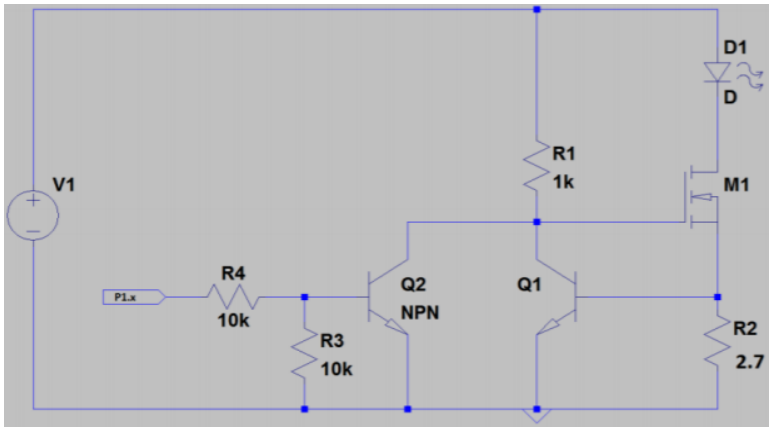

FIGURA 38. CIRCUITO *PWM*.
FONTE: AUTOR (2018).

O *layout* utilizado para confecção da placa são apresentados nas FIGURAS 39 e 40, com e sem os componentes, respectivamente.

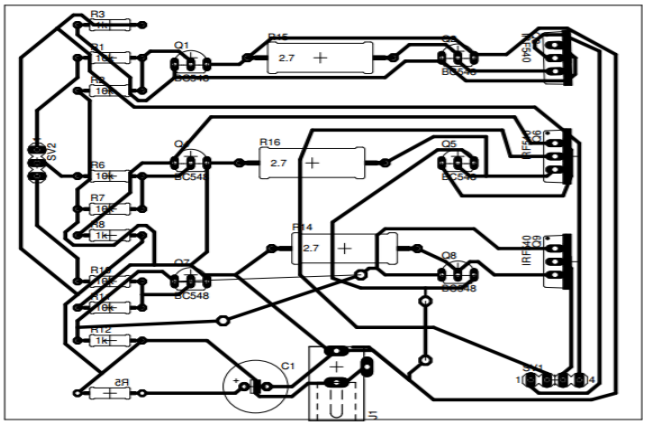

FIGURA 39. *LAYOUT* PLACA *RGB* COM OS COMPONENTES.
FONTE: AUTOR (2018).

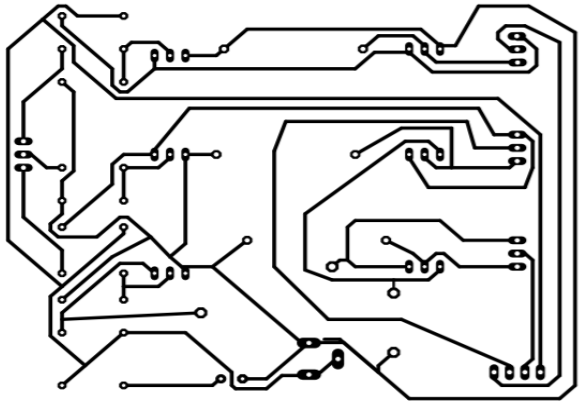

FIGURA 40. *LAYOUT* PLACA *RGB* SEM OS COMPONENTES.
FONTE: AUTOR (2018).

O *LED* (FIGURA 41) utilizado nesse módulo é um *LED* de potência média de 3 Watts e implementa três *LEDs* no mesmo encapsulamento, um vermelho, um verde e um azul. As 3 cores combinadas são capazes de fornecer uma grande gama de variedades e tons de cores.

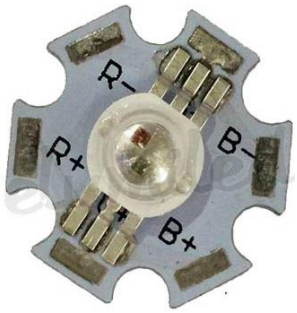

FIGURA 41. *LED RGB*
FONTE: USINAINFO (2016).

3.5. ESPTOUCH

Para o desenvolvimento do protocolo *ESP-TOUCH* nesse projeto de *IOT* é necessário a alteração do código no dispositivo (*firmware*) e a alteração no código do aplicativo *mobile*.

As alterações do firmware (FIGURA 42) do dispositivo são provenientes da exclusão das instruções que antes faziam a conexão com a rede de maneira forçada, ou seja, escrevendo o SSID e o password da rede diretamente no código do dispositivo, o que claramente não é bom para aplicações para o publico geral ou leigo. Então é adicionado as instruções responsáveis pela configuração inteligente do dispositivo.

```
void setupWiFi(){
  int cnt = 0;

  WiFi.mode(WIFI_STA);

  while(WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
    if(cnt++ >= 10){
       WiFi.beginSmartConfig();
       while(1){
           delay(1000);
           if(WiFi.smartConfigDone()){
             Serial.println("SmartConfig Success");
             break;
           }
       }
    }
  }

  Serial.println("------------");
  Serial.println("-----wifi info:------");

  WiFi.printDiag(Serial);

  // Start the server
  UDP.begin(5000);
  Serial.println("UDP Server started on port 5000, module ip is:");

  // Print the IP address
  Serial.println(WiFi.localIP());

}
```

FIGURA 42. NOVO CÓDIGO PARA CONFIGURAÇÃO WI-FI DO DISPOSITIVO. LADO DO ESP
FONTE: AUTOR (2019).

Para a configuração e adaptação do aplicativo Android para o uso do protocolo *ESPTOUCH*, foi necessário analisar e reutilizar parte do código fonte do aplicativo fornecido pela fabricante. Alguns pacotes e algumas classes devem ser incorporadas ao aplicativo do projeto para que ao iniciar o processo de adição de um novo módulo, será inserido automaticamente caso o usuário já esteja conectado a rede no momento da abertura da *activity* e outro campo para que ele possa adicionar a senha da rede *Wi-Fi* caso seja uma rede protegida.

A primeira versão da tela de adição de um novo módulo com esses campos adicionados, conforme mostra a FIGURA 43.

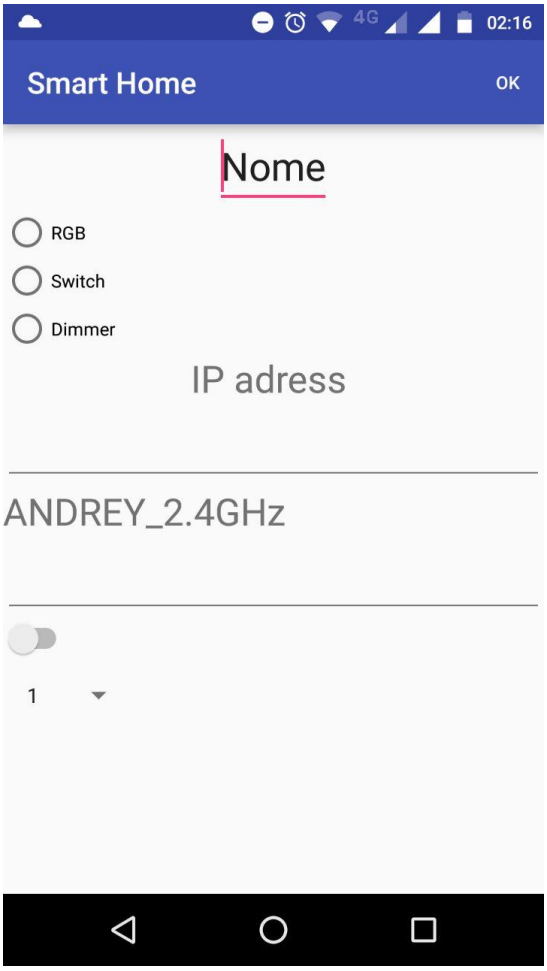

FIGURA 43. TELA APLICATIVO PARA CONECTAR ESP 2017/2.
FONTE: AUTOR (2018).

A senha da rede deve ser inserida no campo logo abaixo ao que esta selecionada a rede *Wi-Fi*.

Após a inserção da senha o usuário apenas deve clicar no botão ok no canto superior direito e o processo de configuração irá se iniciar.

Esse processo inicia uma nova *thread* no sistema Android e enquanto ela está sendo executada.

Com o sucesso do processo de configuração, nessa caixa de dialogo irá aparecer o *IP* do módulo conectado, o sistema irá salvar os dados referentes ao módulo, bem como seu *IP* no banco de dados *SQLite* do Android, a caixa de dialogo será fechada junto com a *'activity'* de adição, retornando assim à lista dos módulos já atualizada com o novo módulo que foi adicionado ao sistema.

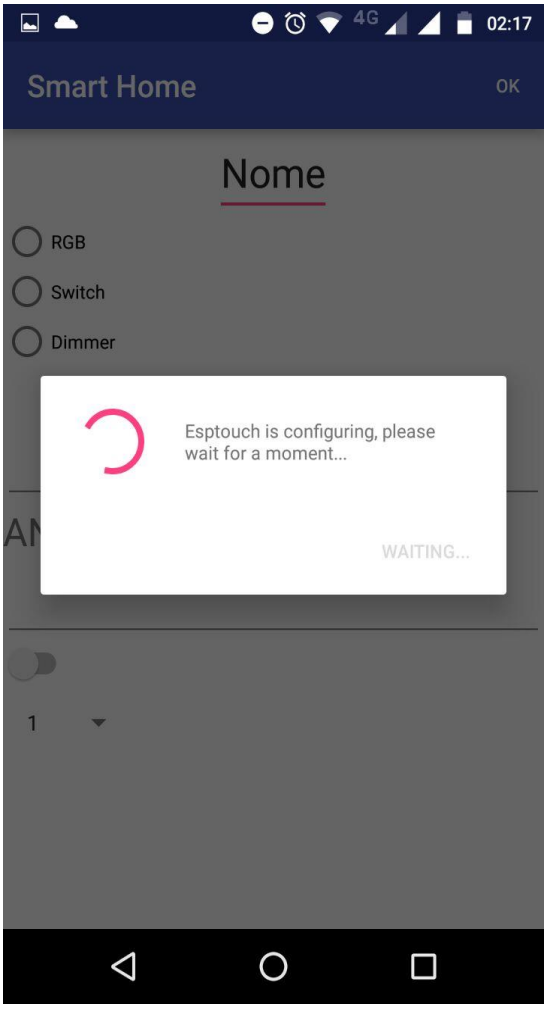

FIGURA 44. ATUALIZAÇÃO DA TELA APLICATIVO PARA CONECTAR ESP 2017/2.
FONTE: AUTOR (2018).

3.5.1. ESP8266 ESP 14

Para dar continuidade ao projeto utilizando o ESP-14 foi necessário realizar a conexão das portas TX e RX através do esquemático que conecta as portas M_PD5 e M_PD6 (FIGURA 45) já que o módulo é a combinação do *STMicro STM8S003F3P6* com o *Expressif ESP8266-EX WiSoC*..

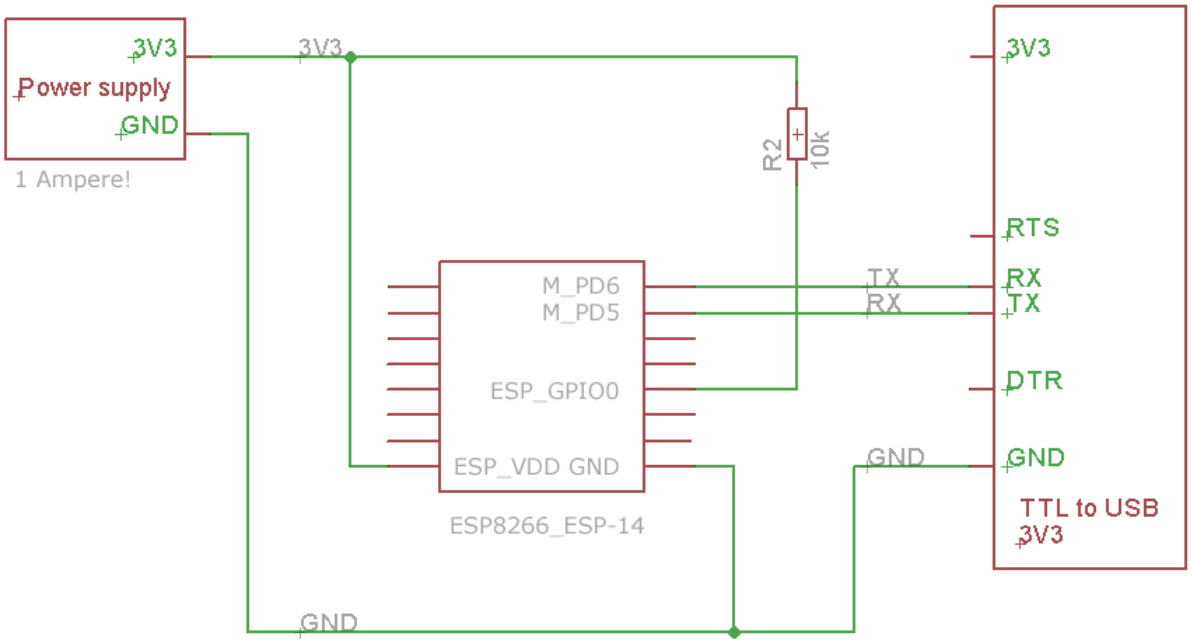

FIGURA 45. ESQUEMÁTICO DE CONEXÃO RX TX ESP-14

FONTE: JONG(2015).

Para a aplicação do módulo, foi necessário implementar uma nova IDE no microcontrolador, para se programar em linguagem de alto nível, um exemplo de programação básica para configurar a rede (FIGURA 46) é apresentado a seguir.

```
1  void setup() {
2    // put your setup code here, to run once:
3    Serial.begin(115200);
4    Serial.println("Test your ESP8266");
5  }
6
7  void loop() {
8    // put your main code here, to run repeatedly:
9    Serial.println("Tick");
10   delay(500);
11   Serial.println("Tack");
12   delay(500);
13 }
```

FIGURA 46. EXEMPLO DE PROGRAMAÇÃO BÁSICA ESP-14

FONTE: JONG(2015).

Para o desenvolvimento dos módulos do projeto, como por exemplo o rele e o circuito de controle do LED RGB também foi necessário uma programação diferente da utilizada anteriormente.

Na parte inicial do código do *firmware* para o circuito de barra de tomadas, o fragmento de código apresentado na FIGURA 47 é responsável pela inicialização do ESP8266. O código define o *buffer* de recebimento dos pacotes, seta os pinos 16, 4 e 5 do ESP em modo de saída para serem usados no acionamento do relé além de inicializar a rotina de configuração automática do *Wi-Fi*.

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; //buffer to hold incoming packet,
WiFiUDP UDP;
int SWITCH1 = 16;
int SWITCH2 = 4;
int SWITCH3 = 5;


void setup() {

  pinMode(SWITCH1, OUTPUT);
  pinMode(SWITCH2, OUTPUT);
  pinMode(SWITCH3, OUTPUT);

  Serial.begin(115200);
  setupWiFi();
 }
```

FIGURA 47. PROGRAMA DE INICIALIZAÇÃO DO ESP-14
FONTE: AUTOR(2019).

A FIGURA 48 apresenta a lógica principal por trás do firmware da barra de tomadas. Para o funcionamento, o microcontrolador recebe uma mensagem proveniente do aplicativo Android e interpreta tal mensagem através do código da figura, caso o ESP tenha recebido um pacote ele irá ler a mensagem que é composta pela primeira letra, um 'S' que indica a mensagem é para o circuito do tipo *switch*.

O segundo caractere são os números 1, 2 ou 3, os quais indicam qual dos reles, ou seja qual das tomadas, devem ser acionadas.

O terceiro item da mensagem indica 0 caso a tarefa seja desligar a tomada e 1 caso a intenção seja ligá-la. Por fim é executada a função *digitalWrite* passando como parâmetro qual *switch* deve ser acionado e se ele deve ser ligado ou

desligado, essa função irá alterar o estado na saída digital do ESP acionando assim o rele.

```
void loop() {
  UDP.parsePacket();
  int packetSize = UDP.parsePacket();
    if (packetSize != 0){
              UDP.read(packetBuffer, UDP_TX_PACKET_MAX_SIZE);
              Serial.println(packetBuffer); // debug printa valor recebido via UDP
                if(packetBuffer[0]=='S'){
                    int thisSwitch = 0;
                    switch(packetBuffer[1]){
                      case '1':
                       thisSwitch = SWITCH1;
                       break;
                      case '2':
                        thisSwitch = SWITCH2;
                       break;
                      case '3':
                        thisSwitch = SWITCH3;
                      break;
                    }
                    bool swBool = false;
                    if(packetBuffer[2]=='1'){
                      swBool=true;
                    }else{
                      swBool=false;
                    }
                    Serial.println(thisSwitch);
                    Serial.println(swBool);
                    digitalWrite(thisSwitch, swBool);
                }
    }
```

FIGURA 48. PROGRAMA ACIONAMENTO DO RELE

FONTE: AUTOR (2019).

No circuito *LED WRGB*, para a parte de inicialização (FIGURA 49), foram escolhidos os pinos 16, 5 ,4 e 0 como saída *PWM* do circuito para acionamento dos *LEDs*, os terminais são configurados como pinos de saída e é iniciada a serial para *debug* e o processo de conexão *Wi-Fi.*

Já para a lógica do acionamento dos *LEDs*, o ESP recebe a mensagem com o conteúdo referente a intensidade luminosa de cada um. A mensagem é uma *String* fixa onde há uma letra correspondente a uma cor seguida de um numero de 3 dígitos de 0 a 255, que corresponde a intensidade luminosa requerida para aquela cor, ou seja, o *duty cycle* do *PWM* para a cor. Um exemplo para uma mensagem onde todos os LEDs estão acessos com o brilho máximo seria

"W255R255G255B255", onde W representa a cor branca, o R a cor vermelha, G o verde e B para o azul.

```
#include <ESP8266WiFi.h>
#include <WiFiUdp.h>

char packetBuffer[UDP_TX_PACKET_MAX_SIZE]; //buffer to hold incoming packet,
WiFiUDP UDP;
int pinWhite = 16;
int pinRed = 5;
int pinGreen = 4;
int pinBlue = 0;


void setup() {
  int cnt = 0;

  pinMode(pinWhite, OUTPUT);
  pinMode(pinRed, OUTPUT);
  pinMode(pinGreen, OUTPUT);
  pinMode(pinBlue, OUTPUT);

  Serial.begin(115200);
  setupWiFi();
}
```

FIGURA 49. PROGRAMA INICIALIZAÇÃO DO LED.

FONTE: AUTOR (2019).

```
void loop() {
  UDP.parsePacket();
  int packetSize = UDP.parsePacket();
    if (packetSize != 0){
              UDP.read(packetBuffer, UDP_TX_PACKET_MAX_SIZE);
              Serial.println(packetBuffer); // debug printa valor recebido via UDP
              if(packetBuffer[0]=='W'){

                  double white = ((packetBuffer[1] - 48)*100 + (packetBuffer[2]-48)*10 +  (packetBuffer[3]-48)) -100;
                  double red =   ((packetBuffer[5] - 48)*100 + (packetBuffer[6]-48)*10 +  (packetBuffer[7]-48)) -100;
                  double green = ((packetBuffer[9] - 48)*100 + (packetBuffer[10]-48)*10 + (packetBuffer[11]-48))-100;
                  double blue =  ((packetBuffer[13] -48)*100 + (packetBuffer[14]-48)*10 + (packetBuffer[15]-48))-100;
                  Serial.println(white);
                  Serial.println(red);
                  Serial.println(green);
                  Serial.println(blue);
                  analogWrite(pinWhite, white);
                  analogWrite(pinRed, red);
                  analogWrite(pinGreen, green);
                  analogWrite(pinBlue, blue);
              }
        }


    delay(100);
}
```

FIGURA 50. PROGRAMA ACIONAMENTO DO LED.

FONTE: AUTOR (2019).

O código interpreta a mensagem e armazena em diferentes variáveis qual o valor que deve ser atualizado para cada cor. É então utilizada a função *alalogWrite* que forma um sinal de *PWM* na saída indicada pelo primeiro argumento da função e com o *duty cycle* modulado de acordo com o segundo parâmetro da função.

4. RESULTADOS

Foi concluído na primeira fase do trabalho os objetivos esperados para a disciplina. Testes foram realizados e assim foi possível analisar o funcionamento esperado em relação a comunicação do sistema e acionamento de dispositivos. Neste tópico será apresentado o que foi desenvolvido.

4.1. SOFTWARE DO SISTEMA PARA ANDROID

Os resultados a cerca do aplicativo Android vem sendo satisfatórios. O aplicativo é capaz de adicionar diversos módulos diferentes e mostrá-los em uma lista para o usuário com nome e endereço *IP* de cada módulo individualmente. Também foi implementado com sucesso uma barra de progresso que permite a interatividade com os dispositivos.

O aplicativo é capaz de salvar as configurações de cada módulo permanentemente na memória do celular através de um banco de dados interno. O *software* desenvolvido é capaz de enviar pacotes sempre que a barra de progresso é deslocada pelo usuário, esses pacotes são enviados para o endereço *IP* que está configurado para o módulo em questão.

Na tela principal do aplicativo uma lista de módulos que contém um nome e um grau de progresso da barra é exibida. Algumas melhorias visuais foram feitas nesse *layout*, como a cor e o formato do botão para adicionar novos módulos. As barras de progresso também foram modificadas para cores indicativas, amarelo e preto, as quais indicam o percentual ligado e desligado, respectivamente.

A FIGURA 51 apresenta a tela inicial gerada pelo *software* desenvolvido, com suas modificações e determinadas funções.

A FIGURA 52 apresenta a tela responsável pela adição de novos módulos ao sistema. Esta tela é aberta quando o botão vermelho do canto inferior direito é pressionado. É neste lugar onde é configurado o nome dado ao módulo, a posição inicial da barra de progresso e o *IP* do módulo que esta sendo adicionado.

Para concluir a adição do módulo, basta clicar no menu superior, no botão "OK" e isso irá finalizar essa *activity*, salvar os dados na memoria e retornar para lista de módulos.

Esta *activity* não foi tratada visualmente para fins de comparação com o novo visual adotado nos módulos da tela principal.

O sistema para o Android do projeto está em constante desenvolvimento e na segunda etapa foram realizadas algumas implementações no software de controle do sistema a fim de otimizar cada vez mais o aplicativo desenvolvido.

FIGURA 51. TELA INICIAL DO APLICATIVO DESENVOLVIDO.

FONTE: AUTOR.

FIGURA 52. TELA DE ADIÇÃO DE MÓDULO DO APLICATIVO.

FONTE: AUTOR.

A FIGURA 53 e FIGURA 54 apresentam as modificações realizadas da interface do aplicativo.

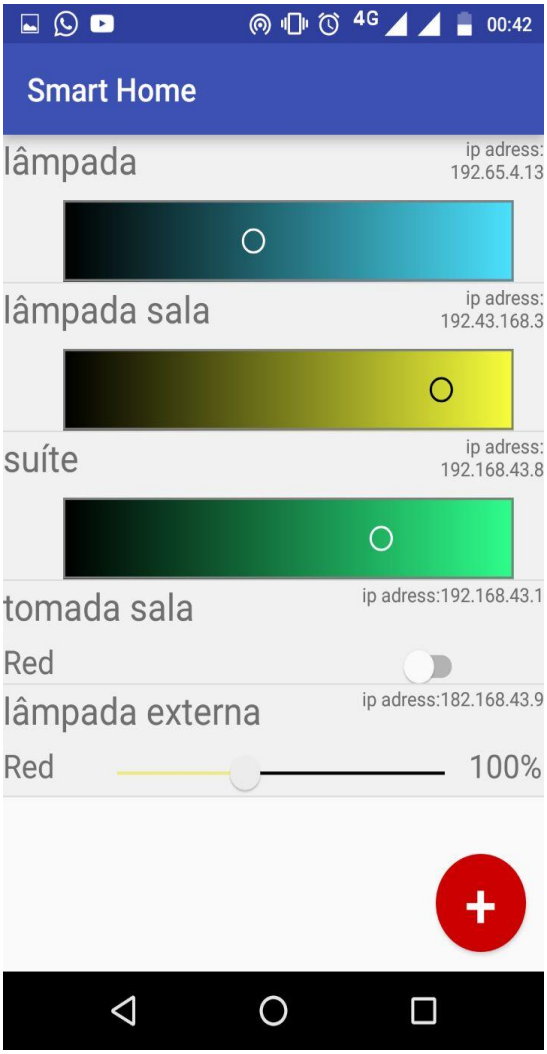

FIGURA 53. MODIFICAÇÃO DO APLICATIVO ANDROID 2017/2.
FONTE: AUTOR (2019).

Na FIGURA 54 observa-se o layout de cada módulo na lista de módulos adicionados do sistema. O primeiro é referente ao módulo do *LED WRGB* que implementa a nova *view* para o controle das cores. Anteriormente era utilizado quatro barras de rolagem e para melhorar a experiência do usuário, agora é possível controlar a intensidade do led através da *ValueView* implementada dentro da lista de módulos. Além disso, uma nova tela foi adicionada para que o usuário tenha acesso a *HueSatView* (view referente a palheta de cores) além de outras configurações mais específicas para o módulo. Um print da tela nova de configurações é apresentado na FIGURA 54. O segundo da lista é o módulo *switch* que implementa

uma chave liga e desliga correspondente ao relé. E por fim o módulo *dimmer* que implementa uma barra de rolagem, já que há apenas um *PWM* para controlar a luminosidade da lâmpada.

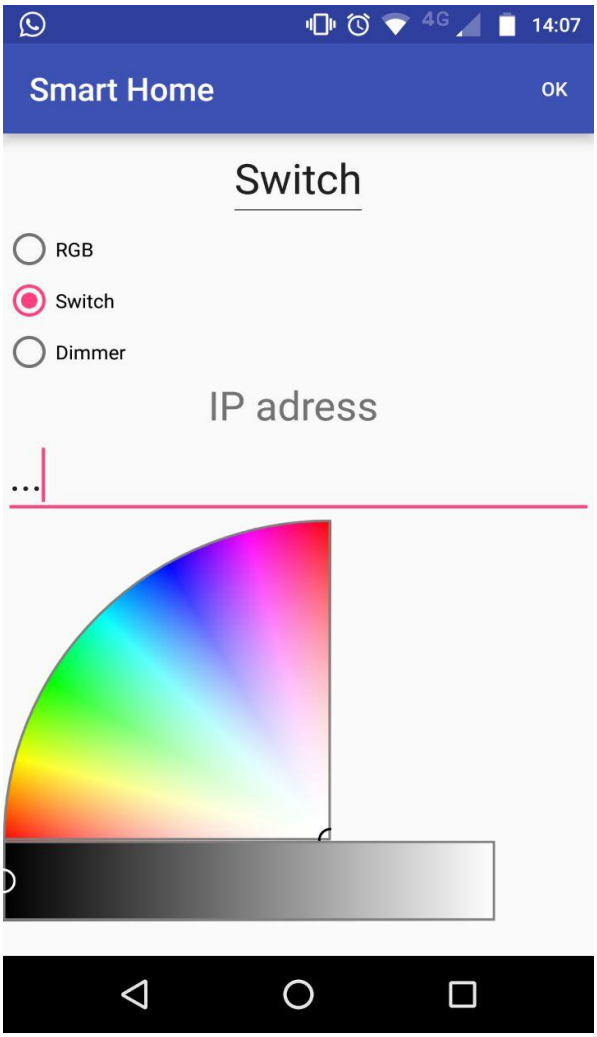

FIGURA 54. MODIFICAÇÃO DO APLICATIVO ANDROID 2017/1 - SWITCH.
FONTE: AUTOR (2019).

A FIGURA 55 apresenta a opção de escolher o módulo que se deseja adicionar, bem como seu endereço *IP* e o nome que o usuário dará ao módulo.

Está em processo de desenvolvimento uma interface que facilite a escolha da cor final no *LED*.

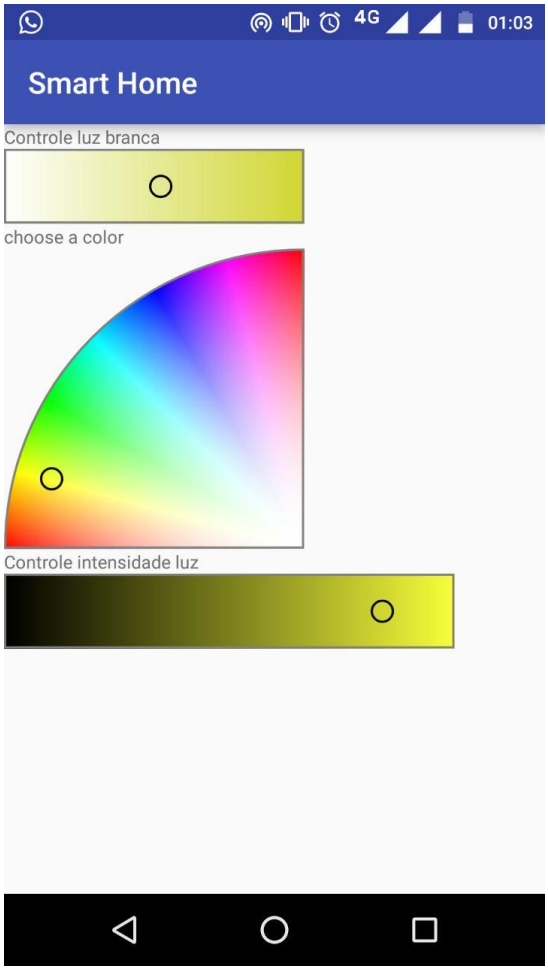

FIGURA 55. MODIFICAÇÃO DO APLICATIVO ANDROID 2017/1 - SWITCH.
FONTE: AUTOR (2019).

4.2. REDE Wi-Fi MQTT COM O ESP8266

A rede para a implementação do sistema já foi configurada e testada utilizando um servidor *broker MQTT*. Após os testes de recebimento do funcionamento, foi possível alterar a taxa de transmissão da rede do ESP para a comunicação com dispositivo de processamento.

A rede desenvolvida está enviando e recebendo dados de informações escolhida pelo usuário, via o software desenvolvido. Ela também é capaz de interpretar o *servidor MQTT,* pelo *software* desenvolvido em Java. Após o envio da informação pelo aplicativo, a o módulo *Wi-Fi* recebe o pacote de dados de comunicação e repassa para o microcontrolador para acionar um atuador.

As configurações da rede foi desenvolvida em linguagem de programação *C*++, via comunicação serial, utilizando a placa *FTDI* para enviar o código ao módulo *Wi-Fi* ESP8266-01.

Para a implementação do código no módulo *Wi-Fi*, se fez necessário atualizar o firmware do mesmo, mudando o *Baud Rate* de transmissão para 112500.

4.3. DISPOSITIVOS DO SISTEMA

A ideia inicial é desenvolver *drivers* em placas de circuito impresso compactas e para isso será realizado estudos a respeito de componentes SMD, ou seja, *Surface Mounted Device*, que são componentes muito menores que os normais.

4.3.1. Módulo Relé

Nesta esta etapa foi desenvolvido um módulo relé para que o sistema de comunicação sem fio já desenvolvido seja capaz de acionar ou desligar equipamentos eletrônicos da residência.

Como já apresentado brevemente no capítulo anterior, um circuito de um relé, ele funciona como uma chave e utiliza um transistor para amplificar a corrente de entrada e um diodo para quando o indutor de acionamento estiver carregado ele se descarregue por ele próprio, afim de não gerar uma corrente reversa e queimar o transistor.

Para elaborar o esquemático do circuito, foi utilizado o *software* de desenvolvimento de *layout* Proteus, onde é possível adicionar componentes livremente e também elaborar o microcontrolador e o dispositivo de comunicação sem fio que está sendo utilizado neste projeto.

A FIGURA 67 apresenta o esquemático do circuito desenvolvido no *software* Proteus.

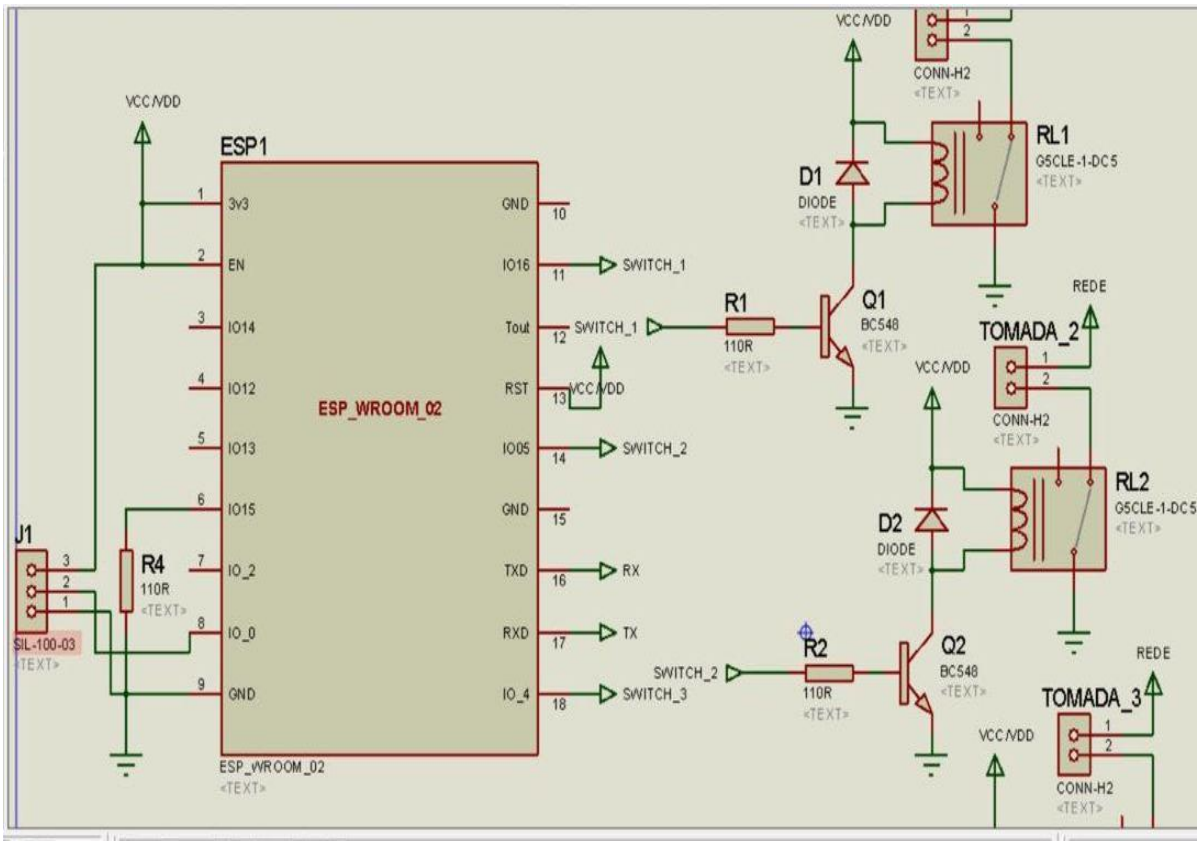

FIGURA 56. PROJETO DE LAYOUT DO DRIVER RELÉ.
FONTE: AUTOR (2019).

O circuito responsável pelo controle do driver relé é bastante simples, há um transistor que permite o fluxo de corrente sobre o indutor do relé quando a saída correspondente do microcontrolador está em nível lógico alto e corta o fluxo de corrente quando a saída do ESP esta em zero. Além disso, há um indutor em paralelo ao acionamento do relé na função de diodo de roda livre, e um conector para ser ligado à um plug de tomada.

Os pinos que são utilizados pelo ESP estão devidamente conectados ao potencial negativo ou positivo, conforme a necessidade do mesmo.

O circuito de acionamento do relé é repetido três vezes para que o módulo funcione como um multiplicador de tomadas inteligente.

Para o desenvolvimento da placa de circuito impresso, foram realizados estudos de componentes para verificar qual seria de melhor benefício, foi realizado comparação entre componentes SMD e componentes normais, foi realizado estudo a respeito de qual seria a melhor maneira de desenvolver *layout* utilizando componentes SMD e seus pré-requisitos, também foi abordado no capítulo anterior

os cuidados e as maneiras que se pode realizar a colocação dos componentes na placa.

Após todo o estudo realizado, foi desenvolvido o layout e também um projeto de visão 3d para a placa (FIGURAS 57 e 58).

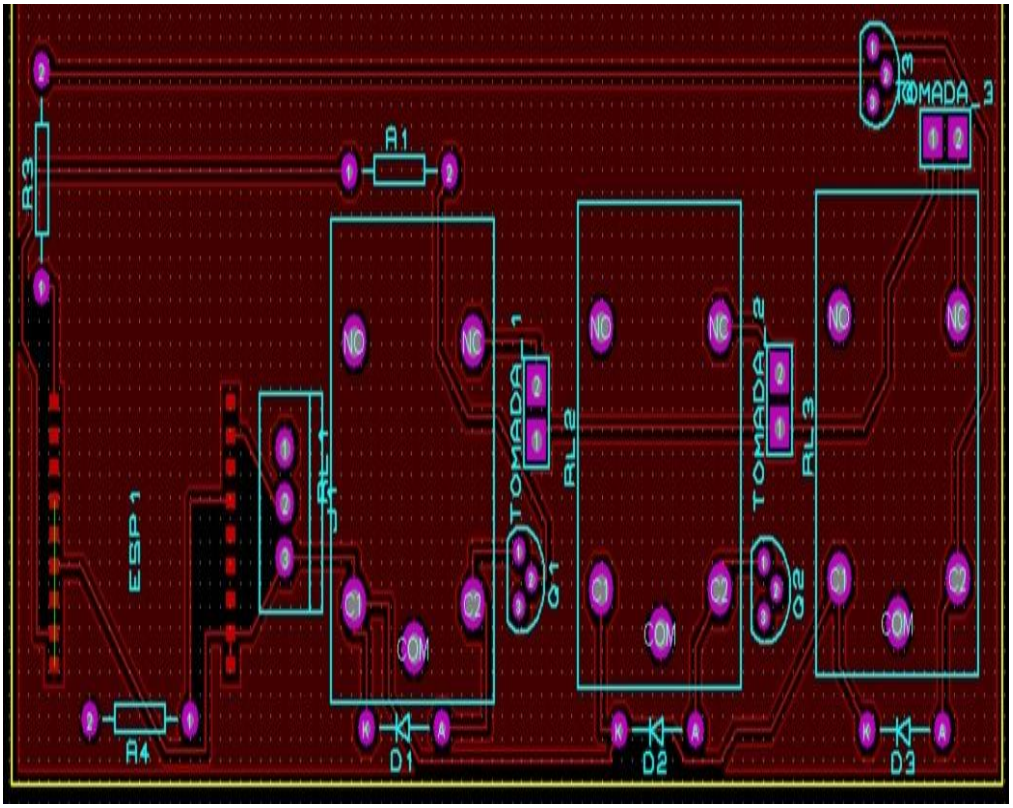

FIGURA 57. PROJETO DE LAYOUT CIRCUITO IMPRESSO RELE.
FONTE: AUTOR (2019).

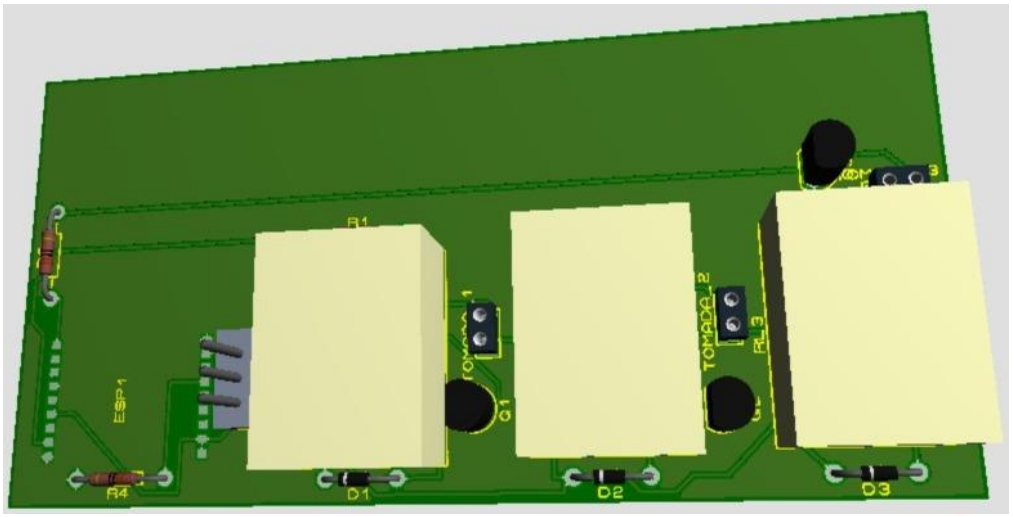

FIGURA 58. VISÃO 3D CIRCUITO IMPRESSO RELE.
FONTE: AUTOR (2019).

4.5.2.1. Simulação

Afim de realizar testes para o circuito desenvolvido e verificar o seu funcionamento correto, foi projetado em um *software* de simulação o respectivo esquemático simplificado do *driver* (FIGURA 59).

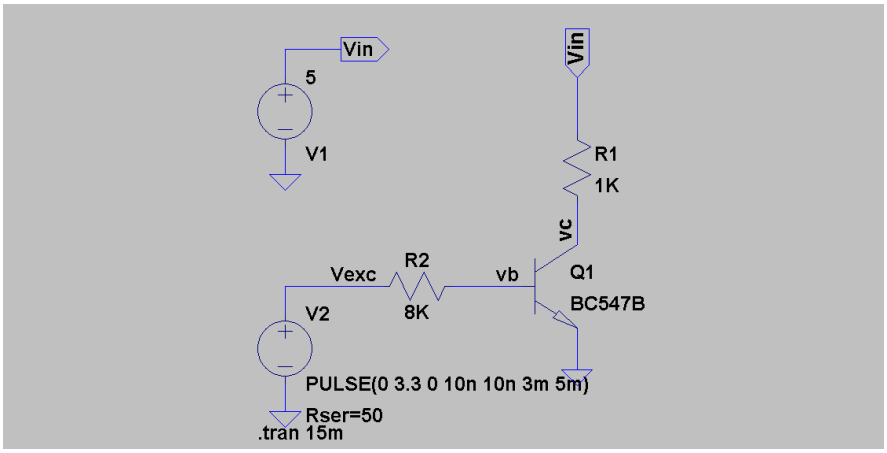

FIGURA 59. CIRCUITO DE SIMULAÇÃO RELE SIMPLIFICADO.
FONTE: AUTOR (2019).

O gráfico gerado (FIGURA 71) a partir da simulação apresenta a saída no resistor conforme a entrada *PWM* do circuito, isso comprova o correto funcionamento do sistema.

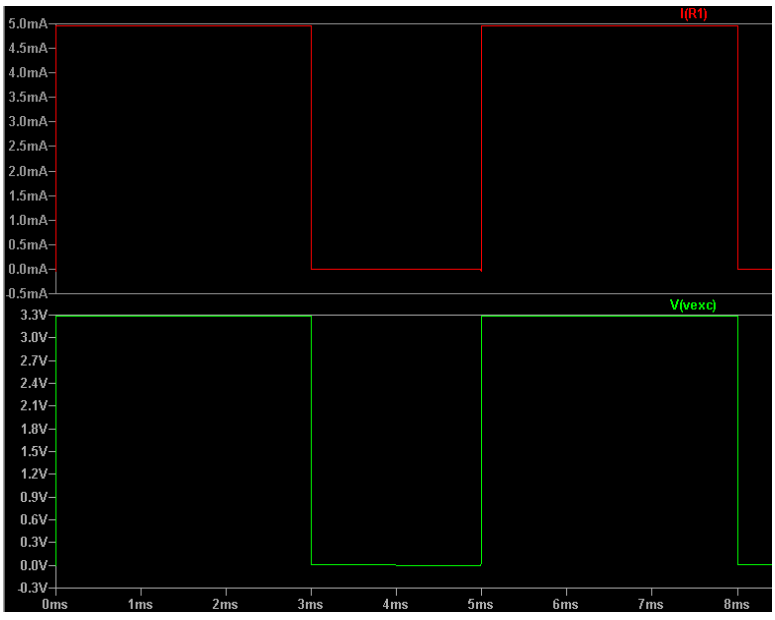

FIGURA 60. GRÁFICO DA SIMULAÇÃO RELE.
FONTE: AUTOR (2019).

A simulação foi desenvolvida utilizando o software LTspice.

4.3.2. Módulo *LED WRGB*

A FIGURA 61 apresenta o circuito *driver* para o *LED WRGB*. Este conjunto é composto por quatro *LEDs*: azul, branco, verde, e vermelho. Cada um deles é controlado individualmente através de um *PWM* e do driver.

Cada lâmpada do encapsulamento consome cerca de 1A para máximo brilho, então é utilizado um transistor MOS-FET para fazer o acionamento de potência do *PWM*. O transistor TBJ e o resistor, que esta conectado sob sua base e emissor, servem para controlar a corrente que irá fluir sobre o *LED*, nesse caso o resistor de 0,7 ohm sob uma tensão de 0,7V da tensão de junção do transistor garante que a corrente passante na malha será de 1A. Este *driver* é repetido quatro vezes, uma para cada *LED*.

Como a tensão de operação do ESP (3,3V) não é suficiente para fazer o acionamento do *LED* de potência, é utilizada outra fonte de tensão de 10V a 15V para os alimentar e um regulador de tensão para 3,3V para alimentar o ESP utilizando a mesma fonte para todo o circuito.

A seguir é apresentado os projetos de layout da placa de circuito impresso (FIGURAS 62, 63 e 64), realizado após estudos feitos a respeito dos componentes, montagem e soldagem da placa.

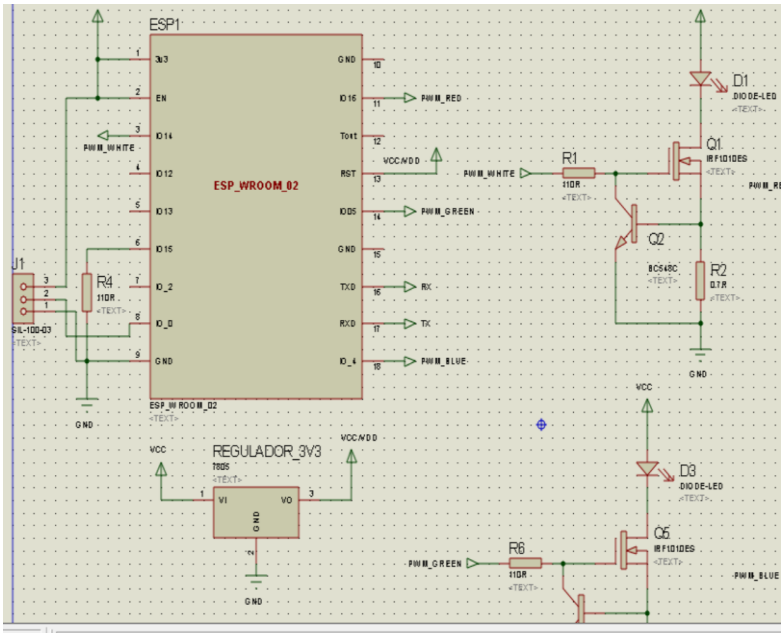

FIGURA 61. PROJETO DE LAYOUT DO DRIVER DO LED WRGB.
FONTE: AUTOR (2019).

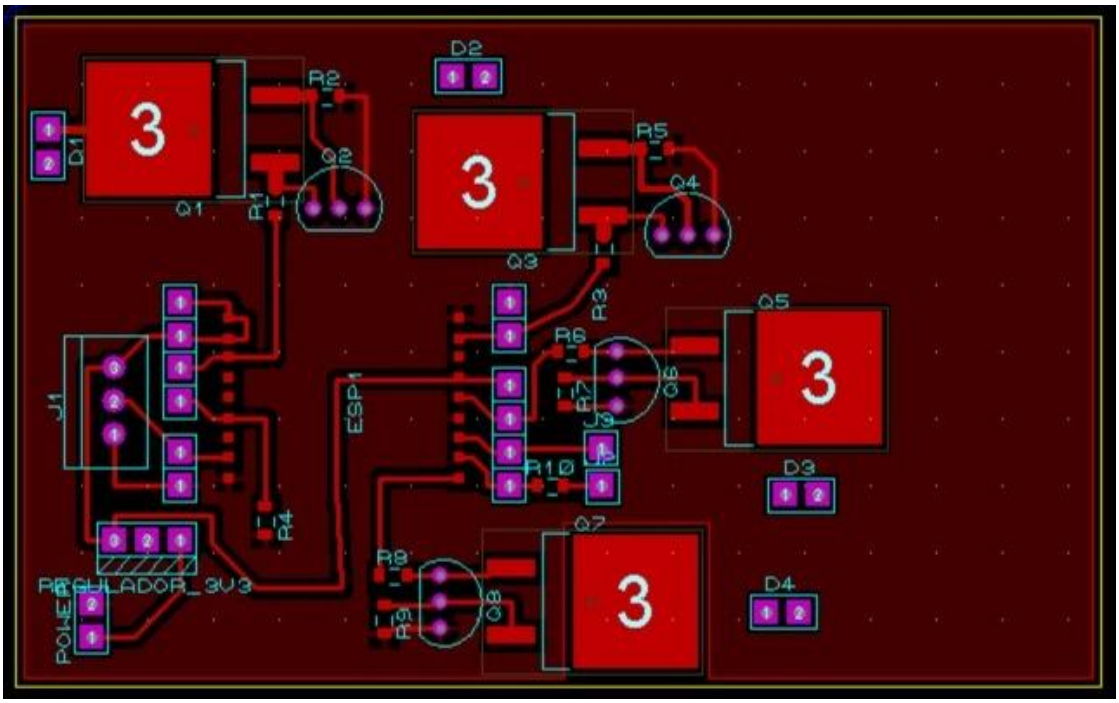

FIGURA 62. PROJETO DE LAYOUT DA PLACA DO LED WRGB.

FONTE: AUTOR (2019).

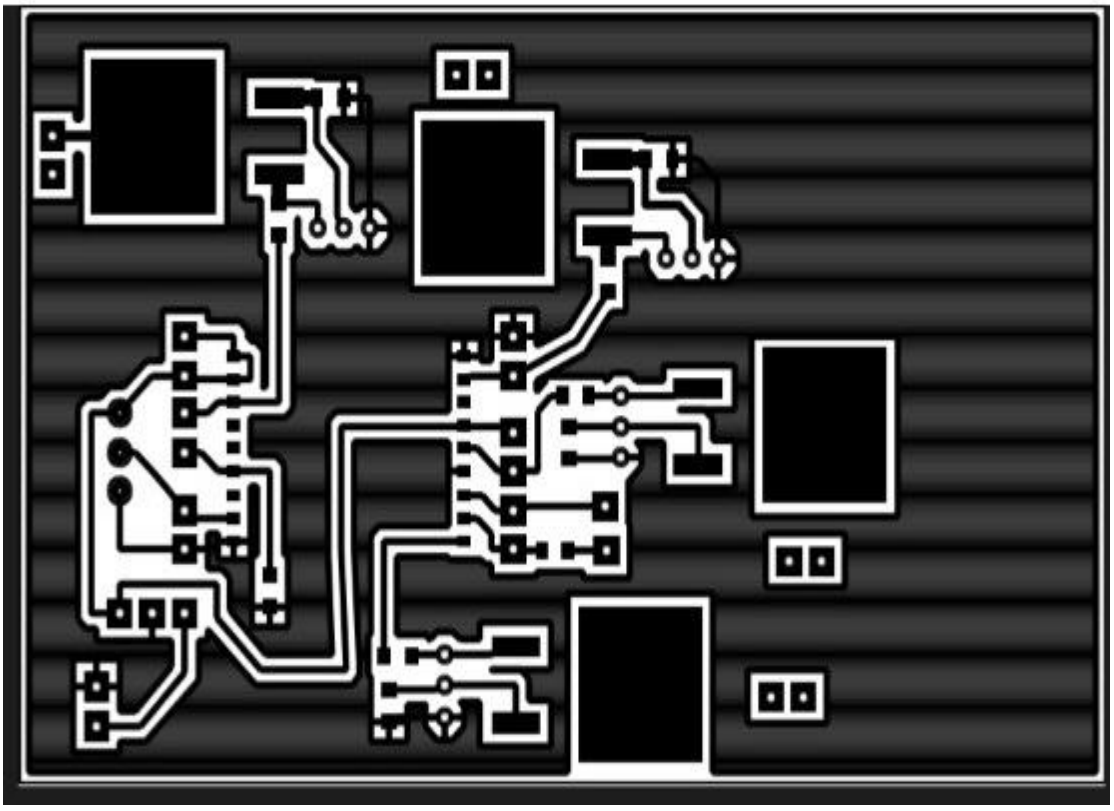

FIGURA 63. PROJETO DE LAYOUT PLACA DO LED WRGB.

FONTE: AUTOR (2019).

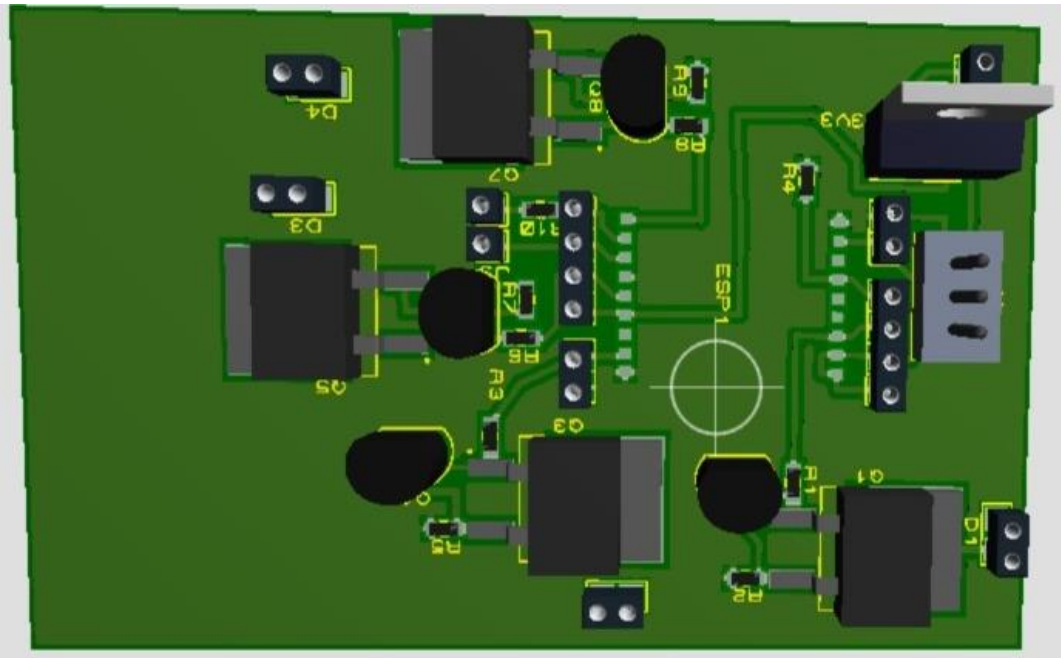

FIGURA 64. VISÃO 3D DA PLACA DO PROJETO DE LED WRGB.
FONTE: AUTOR (2019).

4.5.3.1. Simulação

Afim de realizar testes para o circuito desenvolvido e verificar o seu funcionamento correto, foi projetado em um software de simulação o respectivo esquemático simplificado do *driver* LED WRGB (FIGURA 65).

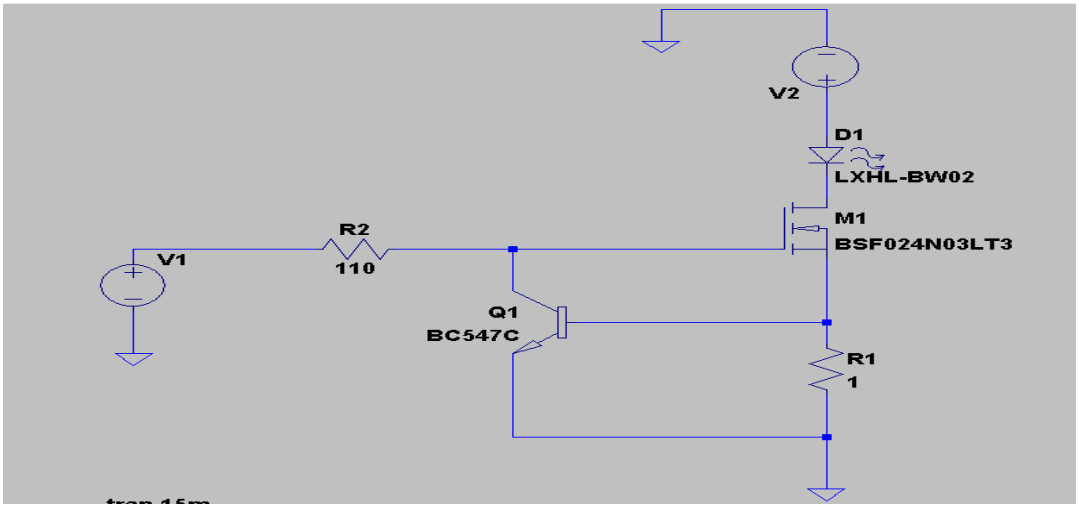

FIGURA 65. CIRCUITO DE SIMULAÇÃO LED WRGB SIMPLIFICADO.
FONTE: AUTOR (2019).

O gráfico gerado (FIGURA 66) a partir da simulação apresenta a saída no resistor (carga) conforme a entrada *PWM* do circuito, isso comprova o correto funcionamento do sistema.

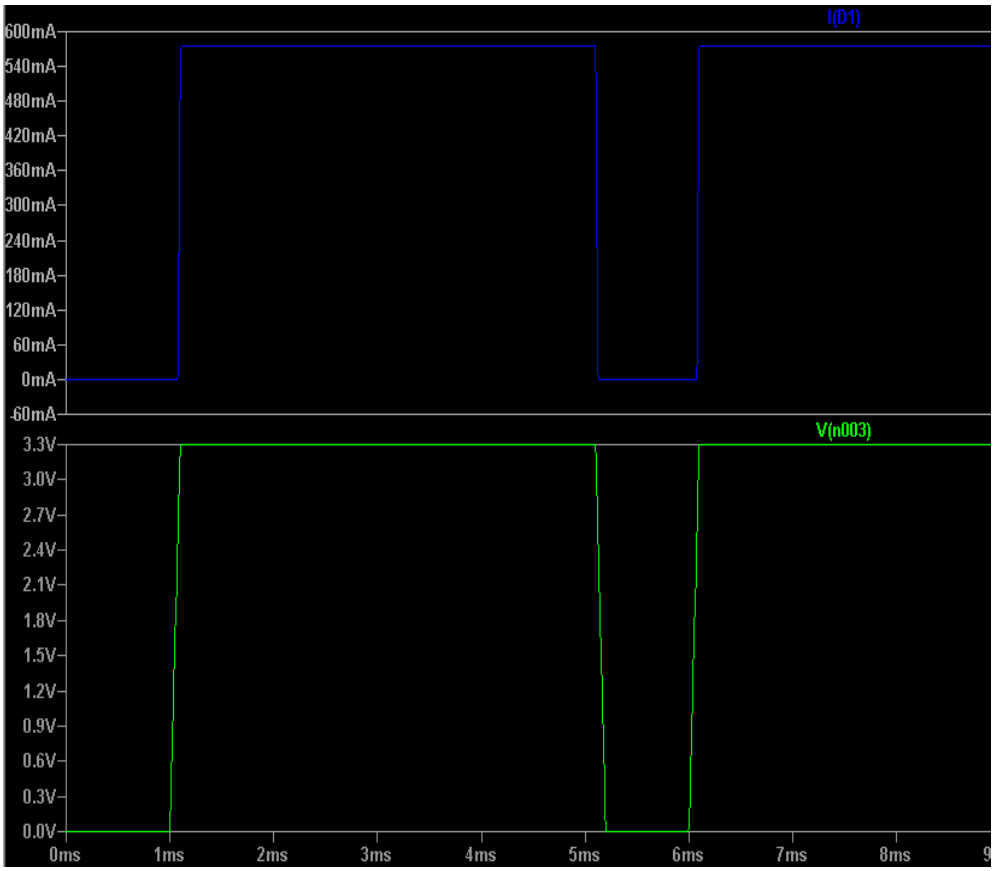

FIGURA 66. GRÁFICO DA SIMULAÇÃO LED WRGB SIMPLIFICADO.
FONTE: AUTOR (2019).

A simulação foi desenvolvida utilizando o software LTspice.

4.5.3.2. Protótipo

O protótipo desenvolvido com a finalidade de testar na prática todo funcionamento do projeto e comprovar conceitos teóricos já apresentados neste trabalho será apresentado neste capítulo.

O protótipo (FIGURA 67) apresentou bons resultados como já era esperado, teve um funcionamento perfeito junto com o ESP e a rede Wi-Fi.

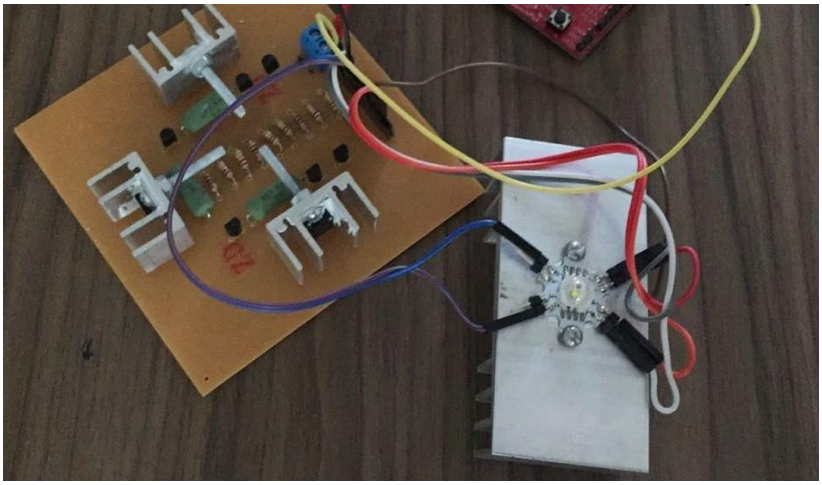

FIGURA 67. PROTÓTIPO LED WRGB.
FONTE: AUTOR (2019).

4.3.3. Módulo *Dimmer*

A FIGURA 68 apresenta o circuito *dimmer* responsável por regular o brilho de lâmpadas comum, o circuito recebe um sinal de *PWM* do microcontrolador ESP e o transmite para o circuito de potência através de um acoplador opto- eletrônico que irá isolar o circuito lógico do circuito de potência da rede.

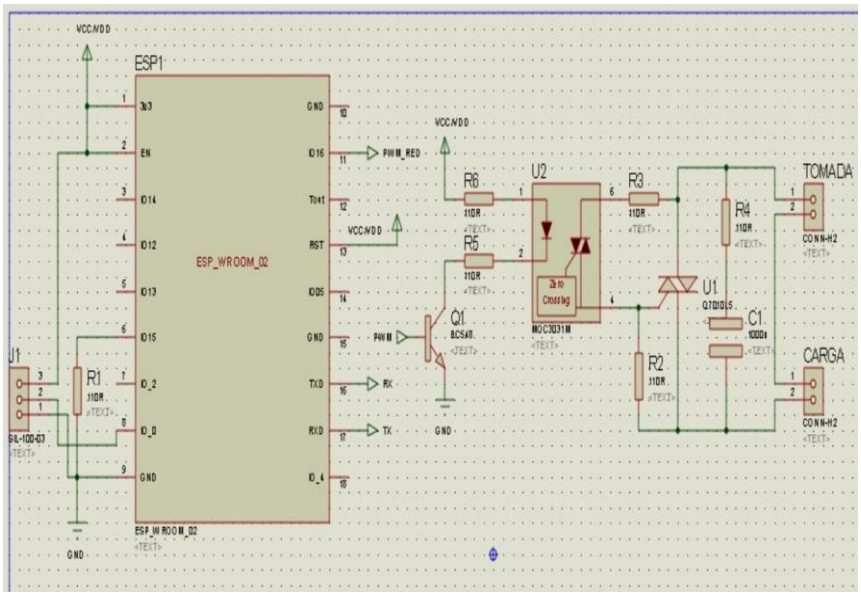

FIGURA 68. PROJETO DE LAYOUT DO DRIVER DO LED WRGB.
FONTE: AUTOR (2019).

O *PWM* no circuito de potência é injetado no *gate* do TRIAC, o qual irá controlar em que ângulo a senoide da rede será cortada, determinando assim quanto de potência será usada para alimentar a lâmpada e quanto será barrado pelo TRIAC, ou seja, controla a tensão RMS da carga, no caso, a lâmpada.

O princípio de funcionamento do circuito é o controle do ângulo de condução de um TRIAC. Disparando-o em diversos pontos do sinal senoidal da rede de energia é possível aplicar a uma carga potências diferentes.

Por exemplo, se o disparo for realizado no início do semiciclo todo ele pode ser conduzido para a carga e ela receberá maior potência. Porém, se o disparo for feito no final do semiciclo, uma pequena parcela da energia será conduzida até a carga que operará com potência reduzida. Pela FIGURA 69 é possível observar o que ocorre.

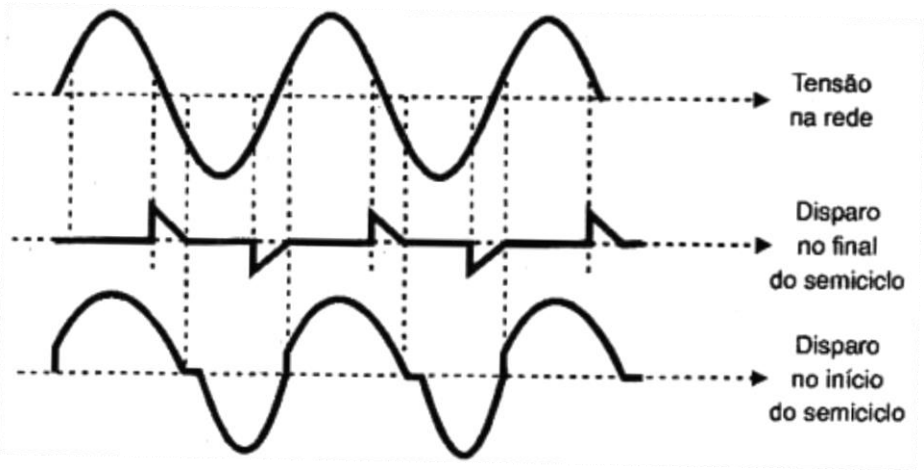

FIGURA 69. ÂNGULO DE DISPARO TRIAC
FONTE: BRAGA (2012).

Para obter o disparo do TRIAC em diversos pontos dos semiciclos da energia da rede o que deve ser feito é usar uma rede RC de retardo onde R é variável.

Conforme a escolha do TRIAC, é possível ter potências diferentes máximas para as cargas controladas. Para isso existem diversas opções utilizando os TRIACs da série TIC da Texas. As opções são:

- TIC116 - 6 amperes;
- TIC226 - 8 amperes;
- TIC236 - 12 amperes;
- TIC246 - 16 ampères.

Após feitos vários estudos a respeito do circuito, componentes e formas de montagem da placa, foi realizado o layout do circuito impresso (FIGURA 70) junto com um projeto 3d da placa (FIGURA 71), para se tornar possível analisar a funcionalidade das dimensões da placa e dos componentes.

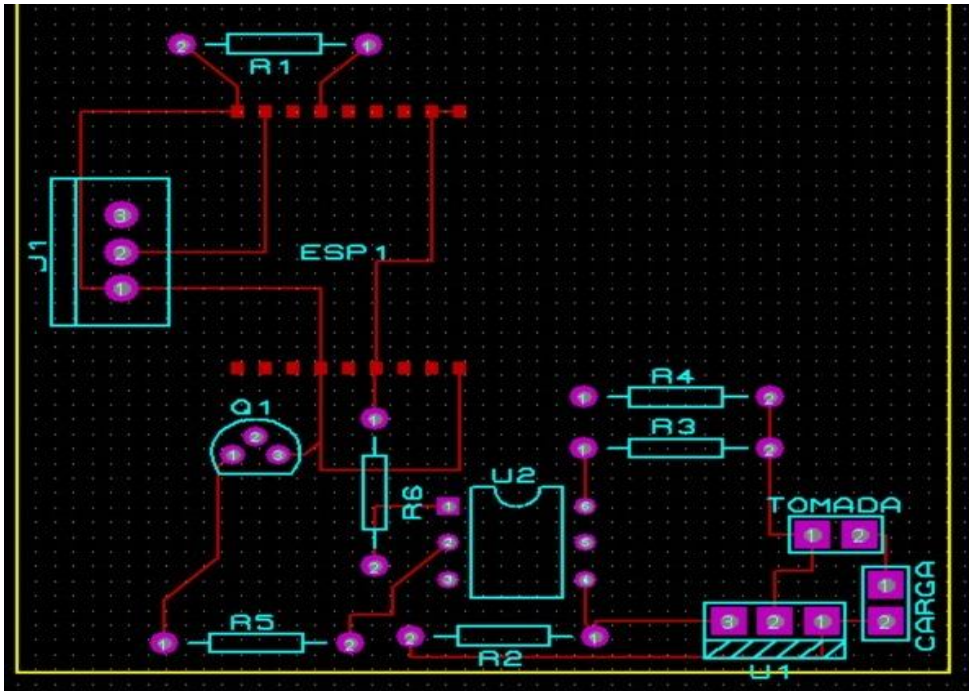

FIGURA 70. LAYOUT PLACA DO PROJETO DO DIMMER.
FONTE: AUTOR (2019).

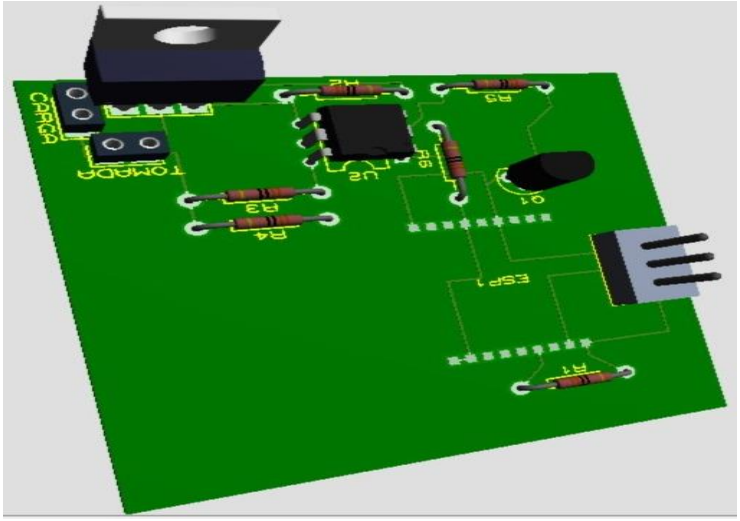

FIGURA 71. VISÃO 3D DA PLACA DO PROJETO DO DIMMER.
FONTE: AUTOR (2019).

4.5.4.1. Simulação

Para realizar os testes para o circuito *dimmer* desenvolvido e verificar o seu funcionamento correto, foi projetado em um *software* de simulação o respectivo esquemático simplificado do driver (FIGURA 72).

O gráfico gerado (FIGURA 73) a partir da simulação apresenta a saída no resistor conforme a entrada *PWM* do circuito, isso comprova o correto funcionamento do sistema.

É possível observar a onda no ângulo de disparo do TRIAC, conforme já visto teoricamente. O circuito desenvolvido está simplificado apenas para regular o ângulo de disparo na parte senoidal positiva, visto que o software utilizado apresenta algumas limitações de componentes e fontes de tensão. O circuito será confeccionado com base no que já foi apresentado e argumentado no relatório.

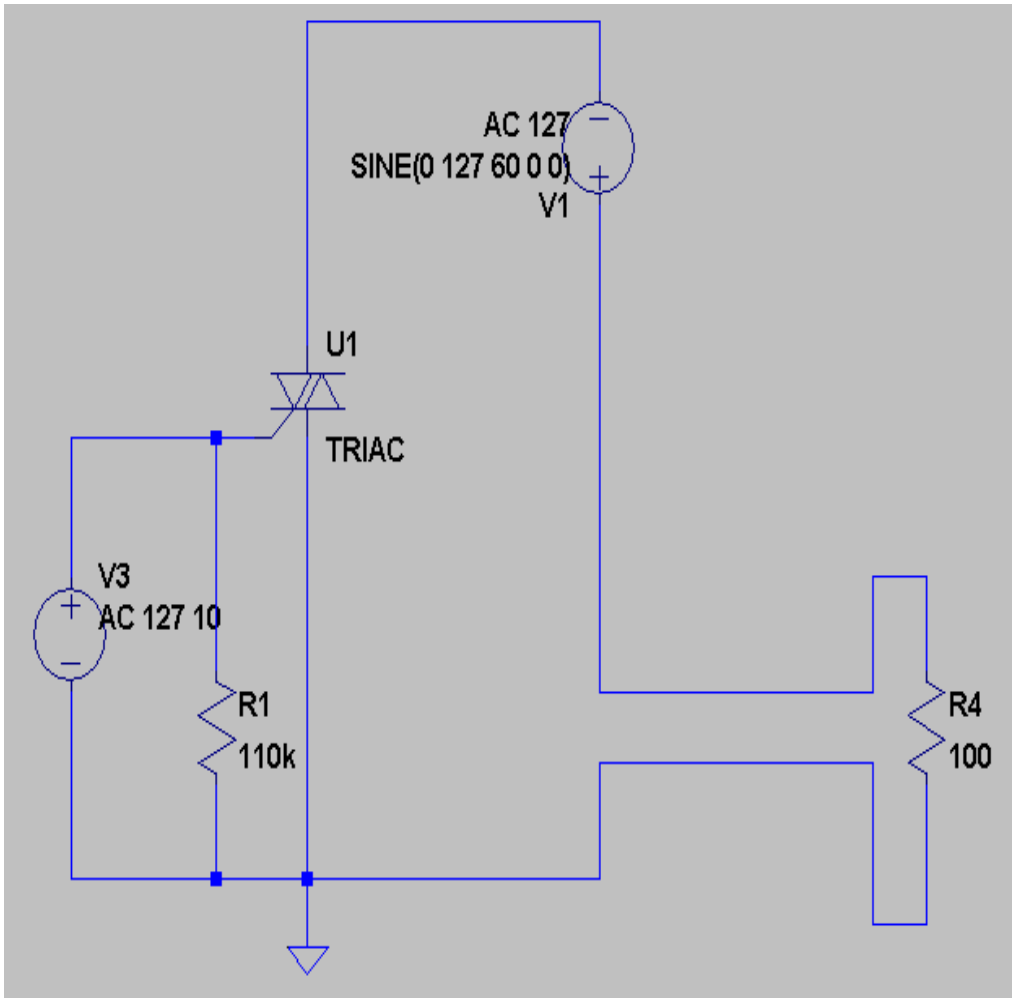

FIGURA 72. ESQUEMÁTICO DE SIMULAÇÃO PROJETO DO DIMMER SIMPLIFICADO.
FONTE: AUTOR (2019).

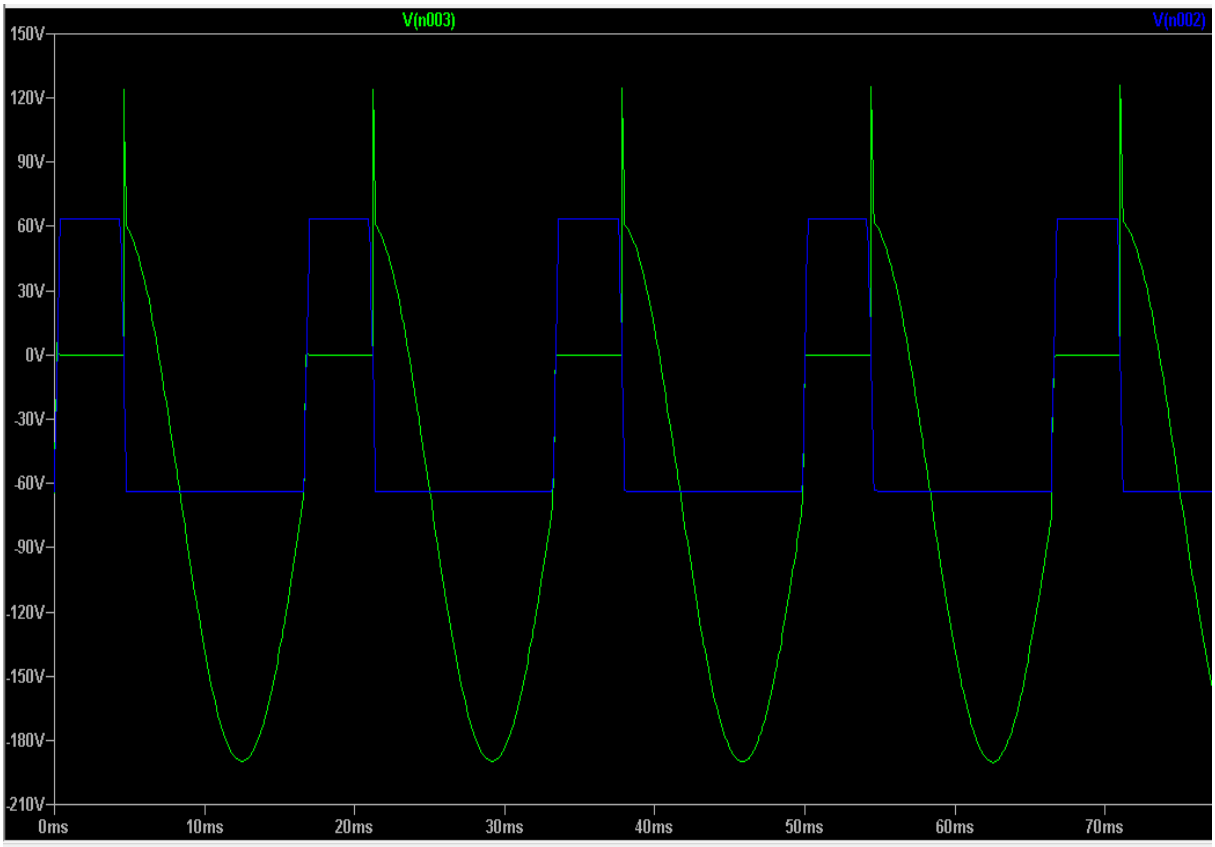

FIGURA 73. GRÁFICO DE SIMULAÇÃO PROJETO DO DIMMER SIMPLIFICADO.
FONTE: AUTOR (2019).

5. CONCLUSÕES PARCIAS

Neste trabalho conseguiu-se abordar conceitos de programação, eletrônica, sistemas operacionais, rede de computadores, sistemas eletrônicos embarcados, linguagem orientada a objeto na área de Engenharia Elétrica.

Para o bom desenvolvimento do trabalho, a equipe dividiu o projeto em três etapas: *software* para o *smartphone*, circuitos do sistema e por fim, a integração de tudo com a nuvem. Para a primeira etapa a equipe reservou 50% do tempo do projeto, para a segunda etapa 30% e para a terceira 20%.

O *software* do projeto desenvolvido em linguagem de programação Java para dispositivos com sistema operacional Android, tem a finalidade de realizar a interface do usuário com o sistema e está operando e realizando suas devidas funções de maneira exata, conforme a equipe planejou desde o princípio.

O aplicativo para *smartphone* está enviando valores através de pacotes de dados para o sistema, o valor a ser enviado é escolhido pelo usuário através da barra de progresso ou de botões *on/off* do *software*

Para segunda etapa do projeto está sendo desenvolvido circuitos para serem implementados no sistema. Estudos foram feitos para que se possa obter resultados eficientes, de baixo custo e de boa funcionalidade. Módulo de controle de tomas e o módulo para o controle e monitoramento de gases e fumaça estão funcionando de maneira perfeita, cumprindo com suas funções.

Para o dispositivo de controle de tomadas, a equipe está desenvolvendo um molde para facilitar a instalação, como um “T” para multiplicação de pontos de energia.

O dispositivo de iluminação está na fase de testes para ser expandido a uma fita de LED ao invés de um LED de potência.

Estudos foram realizados a respeito de desenvolvimento de placa de circuito impresso utilizando componentes SMD, quais seus benefícios em relação a componentes normais e quais suas desvantagens.

Após análise realizada, foi possível desenvolver o desenho da placa já utilizando circuitos SMD, após verificar que para a aplicação do projeto seria benéfico a utilização deste tipo de material. Com o desenvolvimento do desenho, foi possível verificar seu tamanho real e suas dimensões. A dificuldade maior

encontrada para o desenvolvimento da placa é a procura de componentes SMD a venda no mercado e o processo de soldagem na placa, já que eles são componentes muito menores em relação aos componentes normais.

Com relação a integração do sistema com a nuvem, a equipe está trabalhando para implementar a rede *MQTT*. A equipe tentou integrar ao sistema da Amazon, o AWS, porém não foi possível obter sucesso pois o protocolo de segurança não é compatível ao ESP. Agora a equipe está trabalhando para utilizar o servidor da IBM e o CloudMQTT.

A equipe está trabalhando sob a orientação do Professor Doutor Marcos Vinicio Haas Rambo do departamento de Engenharia Elétrica da Universidade Federal do Paraná, que vem ajudando muito desde o princípio.

6. REFERÊNCIAS BIBLIOGRÁFICAS

1 UFPR. **Orientação para normalização de trabalhos acadêmicos: exemplos de referências.** Curitiba, 2013. Disponível em: http://www.portal.ufpr.br/tutoriais_normaliza/referencia_exemplo.pdf. Acesso em 14 de abril 2019.

2 MINATEL, PEDRO. **ESP8266: O guia básico do hardware**. Disponível em: <http://pedrominatel.com.br/eletronica/esp8266-o-guia-basico-de-hardware/> Acesso em 29 jun. 2018.

3 AUTOMATICHOUSE. **Automação residencial.** Disponível em: <http://www.automatichouse.com.br/automacao-residencial/o-que-e-automacao-residencial>. Acesso em 18 out. 2018.

4 LENZ, ANDRÉ. **A Pré-História da Domótica.** Disponível em <http://info-domotica.blogspot.com.br/>. Acesso em 22 out. 2018.

5 BORTOLUZZI, MATIAS. **Histórico da Automação Residencial**. Disponível em <http://sraengenharia.blogspot.com.br/?view=classic>. Acessado em 22 out. 2018.

6 PINTO, PEDRO. **Redes – Sabe o que são sockets de comunicação?** Disponível em: <https://pplware.sapo.pt/tutoriais/networking/redes-sabe-o-que-sao-sockets-de-comunicacao-parte-i/>. Acessado em: 05 set. 2018.

7 REBOUÇAS, LUCAS. **Sockets – Conceitos e Definições Básicas**. Disponível em: <http://tencnolife.blogspot.com.br/2012/05/sockets-conceitos-e-definicoes-basicas.html> . Acessado em: 05 set. 2018.

8 NATIONAL INSTRUMENTS. **Construindo Aplicações em Rede com a Biblioteca UDP do LabWindows** Disponível em: <http://www.ni.com/white-paper/6723/pt/> . Acessado em: 05 set. 2018.

9 NASCIMENTO, EDMAR. **Introdução a Redes de Computadores**. Disponível em: <http://www.univasf.edu.br/~edmar.nascimento/redes/redes_20112_aula02.pdf>. Acessado em: 10 set. 2018.

10 CBPF. **O Protocolo TCP**. Disponível em: <http://www.cbpf.br/~sun/pdf/tcp.pdf>. Acessado em 10 set.2018.

11 TACLA, CESAR. **Sockets UDP, TCP e MULTCAST**. Disponível em: <http://www.dainf.ct.utfpr.edu.br/~tacla/JAVAProgParSD/0050-Sockets.pdf>. Acessado em: 15 set. 2018.

12 TUTORIALSPOINT. **Embedded Systems - 8051 Microcontroller**. Disponível em: <https://www.tutorialspoint.com/embedded_systems/es_microcontroller.htm> Acessado em: 15 set. 2018.

13 CAVALCANTE, SAMUEL. **Arquitetura de Microprocessadores e Microcontroladores**. Disponível em: <http://blog.samuelcavalcante.com/wp-content/uploads/2012/03/Aula-3-Arquiteturas-Microcontroladores-e-Microprocessadores.pdf > . Acessado em 20 set. 2018.

14 ELETRONICAPROGRESSIVA. **Microcontroladores – O que são, Para Que Servem e Onde São Usados**. Disponível em: <http://www.eletronicaprogressiva.net/2014/08/Microcontroladores-O-que-sao-Para-que-servem-Onde-sao-usados.html >. Acessado em 20 set. 2018.

15 PALMIERE, SÉRGIO. **CLP Versus Microcontroladores**. Disponível em: <https://www.embarcados.com.br/clp-versus-microcontrolador/ >. Acessado em: 20 set. 2018.

16 ZELENOVSKY, RICARDO E MENCONÇA, ALEXANDRE. **Arquitetura de Microcontroladores Modernos**. Disponível em: <http://www.mzeditora.com.br/artigos/mic_modernos.htm > . Acessado em: 20 set. 2018.

17 RADIOAMADORES. **Microcontroladores**. Disponível em: <http://www.radioamadores.net/files/microcontroladores_pic.pdf > . Acessado em: 20 set. 2018.

18 WEBER, GUSTAVO. **Microcontroladores**. Disponível em: <http://www.joinville.udesc.br/portal/professores/eduardo_henrique/materiais/apostila_micro_do_Gustavo_Weber.pdf > . Acessado em 21 set. 2018.

19 NATIONALINSTRUMENTS. **Conceitos Gerais de Comunicação Serial**. Disponível em:< http://digital.ni.com/public.nsf/allkb/32679C566F4B9700862576A20051FE8F>. Acessado em: 25 set. 2018.

20 BRAGA, NEWTON. **Como Funcionam As UARTs**. Disponível em: <http://newtoncbraga.com.br/index.php/telecomunicacoes/1709-tel006> . Acessado em 25 set. 2018.

21 TEXASINSTRUMENTS. **KeyStone Architecture**. Disponível em: <http://www.ti.com/lit/ug/sprugp1/sprugp1.pdf > . Acessado em 01 nov. 2018.

22 SOUZA, CLEIDSON. **Conceitos de Orientação a Objetos**. Disponível em: <http://www.ufpa.br/cdesouza/teaching/es/3-OO-concepts.pdf> . Acessado em 05 nov. 2018.

23 NOVAELETRONICA. **Circuito de Dimmer Usando Transistor**. Disponível em: <http://blog.novaeletronica.com.br/circuito-de-dimmer-usando-transistor-mosfet/ > . Acessado em 8 nov. 2018.

24 FLOP, FILIPE. **Módulo ESP8266 ESP-01**. Disponível em: <http://www.filipeflop.com/pd-1f55ad-modulo-wifi-esp8266-esp-01.html > . Acessado em 9 nov. 2018.

25 PATECHSOLUTIONS**. How to Interface Relay With MSP430**. Disponível em: < https://www.pantechsolutions.net/microcontroller-boards/relay-interfacing-with-msp430f5529> . Acessado em 10 nov. 2018.

26 GARANTE, ENRICO. **MSP430 Lauchpad Tutorial**. Disponível em: < https://www.embeddedrelated.com/showarticle/420.php> , Acessado em 11 nov. 2018.

27 OLSON, DAVID**. Scientifc Instruments Using The TI MSP430**. Disponível em: < http://mspsci.blogspot.com.br/2012/05/tutorial-18-finishing-uart-transceiver.html>. Acessado em: 13 nov. 2018.

28 REIS, EDGAR. **ESP UART Communication Over Wi-Fi**. Disponível em: < http://www.esp8266.com/viewtopic.php?f=29&t=6840>. Acessado em 10 nov.2018.

29 SPARKFUN. **ESP8266 AT Instruction Set**. Disponível em: < https://cdn.sparkfun.com/assets/learn_tutorials/4/0/3/4A-ESP8266__AT_Instruction_Set__EN_v0.30.pdf>. Acessado em 10 nov. 2018.

30 **DRAW.IO.** Disponível em: < https://www.draw.io/ >. Acessado em 07 dez. 2018.

31 FILIPEFLOP. **PLACA FTDI**. Disponível em: < http://www.filipeflop.com/pd-14690c-placa-ftdi-ft232rl-conversor-usb-serial.html>. Acessado em 02 dez. 2018.

32 ELÉTRICA UFPR. **LISTA DE TCC**. Disponível em: < http://eletrica.ufpr.br/p/tcc:listagem>. Acessado em 03 dez. 2018.

33 EMBEDDED. **HOW TO CALL CCR0 VAL**. Disponível em: < https://www.embeddedrelated.com/showthread/msp430/2482-1.php >. Acessado 10 dez. 2018.

34. TI DESIGN. **SOFTWARE RGB LED CONTROL**. Disponível em: < http://www.ti.com/lit/ug/tidu761/tidu761.pdf>. Acessado em 10 dez. 2018.

35. ELECTRONICA. **Tiristor - Triac**. Disponível em: < https://www.electronica-pt.com/triac-tiristor> Acessado em 01/mai/2019.

3 **Noções básicas sobre componentes eletrônicos**. Disponível em < - http://www.mutcom.no.comunidades.net/>. Acessado 01 mai. 2019.

37.COMPONENTES ELETRÔNICOS - CONCEITOS Disponível em: <https://fbseletronica.wordpress.com/2014/04/29/componentes-eletronicos-conceitos-basicos/>. Acessado em 05 mai. 2019.

38. PROFESSOR DARIO. PERNAS. Disponível em: <http://www.jf.ifsudestemg.edu.br/dario/baixar/pernas.pdf>. Acessado em 05 mai. 2019.

39.ELETRONICA DIARIA. Disponível em: <http://eletronicadidatica.com.br/componentes/tiristor/tiristor.htm> Acessado em 07 mai. 2019.

40.BRAGA. **SCR**. Disponível em: < http://www.newtoncbraga.com.br/index.php/artigos/49-curiosidades/4114-art561.html>. Acessado em 08 mai. 2019.

41. BRAGA. CONTROLE DE POTENCIA. Disponível em: < http://www.newtoncbraga.com.br/index.php/artigos/54-dicas/1025-controle-de-potencia-com-triac-art149.html> Acessado 08 mai. 2019.

42. BRAGA. ARTIGOS E PROGETOS. Disponível em: <http://www.newtoncbraga.com.br/index.php/eletronica/57-artigos-e-projetos/6414-art807>. Acessado 08 mai. 2019.

43.PROGRAMARPICENC. FILTROS. Disponível em: < http://programarpicenc.com/articulos/solucion-de-problemas-y-errores-con-los-microcontroladores-pic/>. Acessado em 09 mai. 2019.

44. EHOW. SNUBBER. Disponível em: <http://www.ehow.com.br/projetar-snubber-como_32305/> Acessado em 10 mai. 2019.

45. API GUIDES. Disponível em: < https://developer.android.com/guide/topics/ui/overview.html?hl=pt-br >. Acessado em 29 mai. 2019

46. DEVELOPER ANDROID. Disponível em: < https://developer.android.com/guide/topics/ui/index.html?hl=pt-br >. Acessado em 03 jun. 2019.

47. CORDEIRO. FILLIPE. Disponível em: http://www.androidpro.com.br/android-views-intro/>. Acessado em 1 set. 2018.

48. DEVELOPER ANDROID. Disponível em: <https://developer.android.com/reference/android/view/View.html>. Acessado em 5 set. 2018.

49. MÓDULO WI-FI ESP 14. Disponível em: < https://www.marinostore.com/iot/329-modulo-wifi-esp8266-esp-14.html>. Acessado em 01 dez. 2018

50. FLASHING AN ESP8266 ESP-14. Disponível em: < https://benjamindejong.com/flashing-an-esp8266-esp-14/>. Acessado em 20 nov. 2018