UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

DANIEL REZENDE SOUZA
FILIPE HIDEKI DE OLIVEIRA ITONAGA
PAULO ARTHUR VARASSIN MOREIRA

SISTEMA DE SOM INTELIGENTE RESIDENCIAL

CURITIBA
2022

DANIEL REZENDE SOUZA
FILIPE HIDEKI DE OLIVEIRA ITONAGA
PAULO ARTHUR VARASSIN MOREIRA

SISTEMA DE SOM INTELIGENTE RESIDENCIAL

SMART HOME SOUND SYSTEM

Trabalho de Conclusão de Curso apresentado ao Departamento Acadêmico de Eletrônica da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do grau de Bacharel em Engenharia Eletrônica. Área de Concentração: Engenharia Eletrônica.

Orientador: Professor Dr. Rafael Eleodoro de Góes

CURITIBA

2022

DANIEL REZENDE SOUZA
FILIPE HIDEKI DE OLIVEIRA ITONAGA
PAULO ARTHUR VARASSIN MOREIRA

SISTEMA DE SOM INTELIGENTE RESIDENCIAL

Trabalho de Conclusão de Curso apresentado ao Departamento Acadêmico de Eletrônica da Universidade Tecnológica Federal do Paraná como requisito parcial para a obtenção do grau de Bacharel em Engenharia Eletrônica. Área de Concentração: Engenharia Eletrônica.

Orientador: Professor Dr. Rafael Eleodoro de Góes

Data de aprovação: 05/Outubro/2022

Rafael Eleodoro de Goes
Doutor
Universidade Tecnológica Federal do Paraná

Luiz Fernando Copetti
Mestre
Universidade Tecnológica Federal do Paraná

Guilherme de Santi Peron
Doutor
Universidade Tecnológica Federal do Paraná

CURITIBA

2022

AGRADECIMENTOS

Agradeço aos meus pais, professores e amigos. Um agradecimento especial à minha família, por aguardar 9 anos de espera. Aos professores Rafael Eleodoro de Góes, Rubens Alexandre de Faria, Jean Simão, Guilherme Peron e Hugo Vieira Neto, por terem me permitido preocupar menos com provas e notas e utilizar o tempo dentro e fora das aulas para focar em obter conhecimento e descobrir novas tecnologias. Agradeço também ao Célio Biassio, responsável por 50% de tudo que eu entrego (menos as falhas, essas eu assumo 100% da culpa).

Daniel Rezende Souza

Agradeço aos meus pais, irmão, professores e amigos. Aos familiares que compreenderam a minha ausência durante a graduação e realização deste trabalho e em especial ao orientador Rafael Eleodoro de Góes, cuja dedicação e conhecimento foram imprescindíveis para a realização deste trabalho.

Filipe Hideki de Oliveira Itonaga

Agradeço a Deus, familiares, professores e amigos que sempre me apoiaram. Sou grato ao professor Rafael Eleodoro de Góes por indicar a direção correta neste trabalho, e a tantos outros que de forma direta ou indiretamente me auxiliaram nesta parte educacional e profissional na minha vida.

Paulo Arthur Varassin Moreira

RESUMO

O projeto tem como objetivo a montagem de um sistema de som controlado por voz. O sistema permite reproduzir diferentes músicas simultaneamente em cômodos diferentes da casa. As músicas não precisam ser armazenadas em memória, sendo baixadas diretamente da rede. A montagem do produto é feita em placa perfurada. O servidor é hospedado em uma placa *Raspberry Pi*, e o cliente, em uma (ou mais) ESP32 anexada a um circuito de tratamento de áudio. O equipamento é destinado a residências e deve ser utilizado com acesso à rede Wi-Fi local.

Palavras-chave: assistente de voz; *Internet of Things*; sistema de som residencial.

ABSTRACT

The project aims to set up a voice-controlled sound system. The system allows you to play different songs simultaneously in different rooms in the house. Songs don't need to be stored locally, they are downloaded directly from the network. The product is assembled on a circuit board. The server is hosted on a Raspberry Pi board and the client on one (or more) ESP32 attached to an audio amplifier circuit. The equipment is intended for homes and must be used with access to the local Wi-Fi network.

Keywords: voice assistant; *Internet of Things*; home sound system.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS, SIGLAS E ACRÔNIMOS

ADC	Analog-to-digital converter
AI	Artificial Intelligence
API	Application Programming Interface
BCLK	Bit Clock Connection
BLE	Bluetooth Low Energy
CPU	Central Processing Unit
DAC	Digital-to-analog Converter
DDR4	Double Data Rate 4
DIN	Data In
DMA	Direct Memory Access
FreeRTOS	Real-time operating system for microcontrollers
GND	Ground
GPIO	General Purpose Input/Output
HDMI	High-Definition Multimedia Interface
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IP	Internet Protocol
I2S	Inter-IC Sound
I/O	Input and Output
JSON	JavaScript Object Notation
NPM	Node Package Manager
OS	Operating System
PCM	Pulse-code modulation
PID	Process ID

PLN	Processamento de linguagem natural
PWM	Pulse Width Modulation
P2P	Peer to Peer
RAM	Random Access Memory
ROM	Read-only memory
STA/AP/STA+AP	Station/Access Point/Station+Access Point
SPI	Serial Peripheral Interface
THD	Total Harmonic Distortion
URL	Uniform Resource Locator
USB	Universal Serial Bus
VIN	Input Voltage
WSEL	Word Select

SUMÁRIO

1 INTRODUÇÃO

Durante as últimas décadas, a evolução da tecnologia proporcionou grandes alterações no modo de viver, e poucos imaginavam as possibilidades tecnológicas que surgiram neste período. A internet se tornou popular e acessível para boa parte da população, a “Internet das Coisas” e a inteligência artificial começaram a fazer parte do cotidiano — além de outras inovações que entraram nas residências com o intuito de facilitar o dia a dia e otimizar o tempo utilizado nas tarefas diárias.

Assim, o conceito de sistema de som residencial foi tornando-se realidade. Inicialmente, com pequenas caixas de som instaladas nos cômodos mais utilizados da casa. Logo após, houve a melhoria na qualidade desses alto-falantes e a possibilidade de fazer a integração entre eles. Então, a ideia de agregar o conceito de “Internet das Coisas” aos aparelhos sonoros se desenvolveu.

A integração entre os aparelhos sonoros com APIs de interpretação de voz proporcionou o controle das ações de reprodução das músicas através de comandos de voz. Ferramentas como *Wit AI* (META, 2021), *IBM Watson* (IBM, 2020) e *Google Dialogflow* (GOOGLE CLOUD, 2021) auxiliam nessas tarefas e surgem como opções para a implementação desses serviços.

Tendo em vista o desenvolvimento desse sistema, o projeto foi desenvolvido de acordo com esses aspectos, na busca por melhorias nessa utilização.

1.1 Justificativa

Considerando que os sistemas de som disponíveis atualmente não contêm o benefício da utilização em vários cômodos da mesma residência, o projeto propõe uma integração completa entre cada espaço da casa e a possibilidade de escolher uma música individualmente.

1.1.1 Casos de uso

Alguns possíveis casos de uso deste sistema são:

Primeiro caso de uso (preparo do café da manhã):

Caso de uso individual:

- Usuário se levanta da cama;
- Liga o servidor do produto;
- Dirige-se à cozinha;
- Pede para tocar uma música direcionada ao cliente da cozinha;
- Música escolhida toca no ambiente da cozinha;
- Usuário pausa a música através do comando de voz;
- Sai da cozinha;
- Desliga o servidor do produto.

Segundo caso de uso (ao término do café da manhã, usuário se direciona ao escritório):

Caso de uso com dois usuários:

- Usuário levanta da cama;
- Liga o servidor do produto;
- Se direciona a cozinha e pede uma música direcionado ao cliente;
- Música tocando no ambiente da cozinha;
- Usuário pausa a música do cliente da cozinha;
- Dirige-se ao escritório;
- Pede para tocar uma música direcionada ao cliente do escritório;
- Música escolhida toca no ambiente do escritório;
- Segundo usuário se dirige à cozinha;
- Pede uma música direcionada ao cliente da cozinha;
- Música escolhida toca no ambiente da cozinha;
- Ambos os usuários se entretêm com suas músicas em seus respectivos ambientes.

Caso de uso com três usuários:

- Usuário levanta da cama
- Liga o servidor do produto
- Usuário se direciona ao escritório

- Pede para tocar uma música direcionada ao cliente do escritório;
- Música escolhida toca no ambiente do escritório;
- Segundo usuário se dirige à cozinha;
- Pede uma música direcionada ao cliente da cozinha;
- Música escolhida toca no ambiente da cozinha;
- Terceiro usuário se dirige ao banheiro;
- Pede para tocar uma música direcionada ao cliente do banheiro;
- Música escolhida toca no ambiente do banheiro;
- Os três usuários se entretêm com suas músicas em seus respectivos ambientes.

1.2 Objetivos

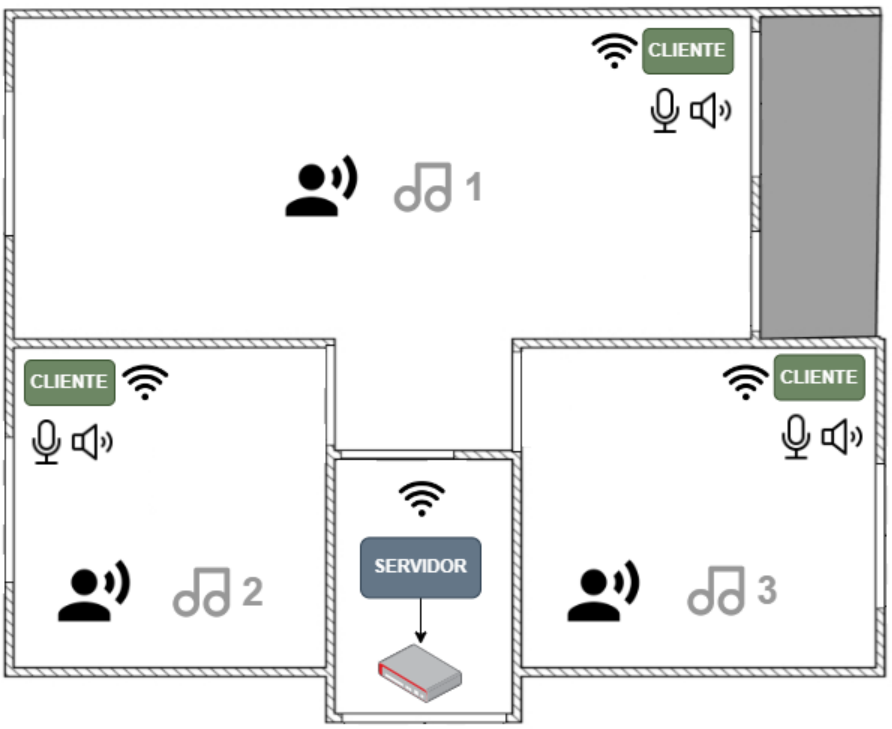
O objetivo do projeto foi o desenvolvimento de um sistema de som residencial que utilizasse o conceito de “Internet das Coisas”, possibilitando saídas e entradas de áudio diferentes em cada cômodo. Nesse sistema, a música reproduzida na saída de áudio é solicitada através de um comando de voz.

Para alcançar esse objetivo, desenvolveu-se um projeto com duas aplicações que rodam no mesmo minicomputador: uma responsável por tratar os dados enviados e recebidos dos clientes (localizados nos cômodos) e outra responsável pela comunicação com a inteligência artificial e fonte de conteúdo.

1.3 Diagrama geral

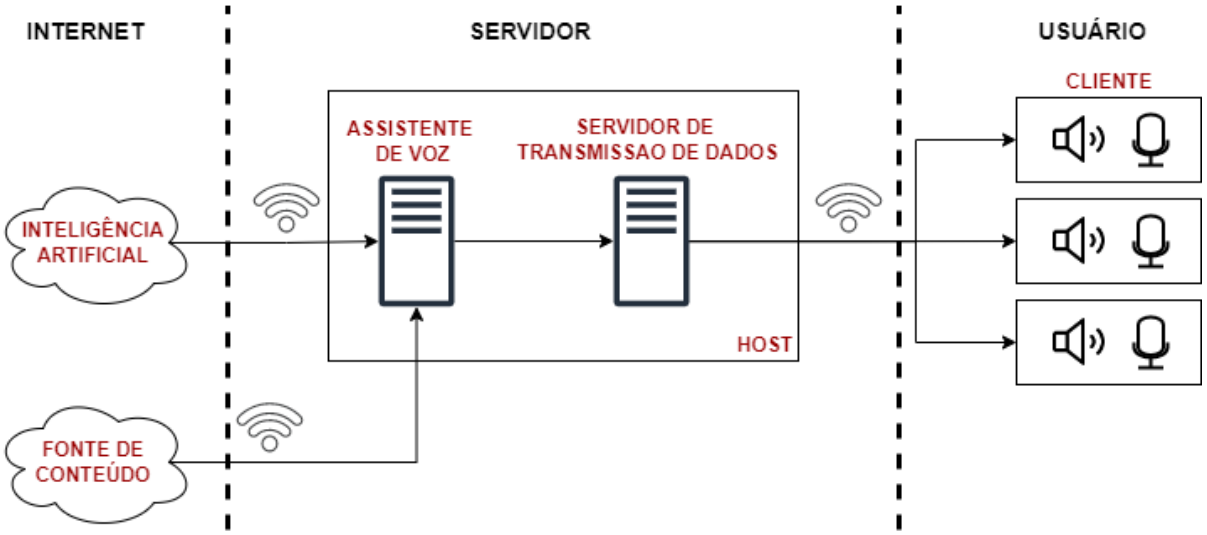
Na figura 1 está demonstrado o uso e na figura 2 o diagrama genérico da solução proposta.

Figura 1 – Diagrama de uso da solução proposta



Fonte: Autoria própria (2022).

Figura 2 – Diagrama genérico da solução proposta



Fonte: Autoria própria (2022).

1.4 Principais componentes

Após estudo, definiu-se qual componente seria usado para cada entidade do modelo proposto. Na figura 3 serão apresentados os componentes (e ferramentas) escolhidos.

A primeira decisão foi a de qual serviço de inteligência artificial seria utilizado. No mercado, existem várias opções, como *Google Dialogflow*, *Botpress* (2021), *IBM Watson* , *Mycroft AI* (2020) e *WIT AI*. Por motivos de custo, as duas opções mais viáveis seriam *Mycroft AI* e *Wit AI*. O critério de desempate que levou à escolha da *Wit AI* foi a usabilidade, pois a *Mycroft AI* é preferencialmente usada com *Python* (ROSSUM, 2020), enquanto a *Wit AI* utiliza preferencialmente *JavaScript* (PLURALSIGHT, 2020), linguagem que já conhecemos. Outras opções existem, mas o material disponível online a respeito dessas ferramentas é consideravelmente maior nas linguagens indicadas. A própria interface da *Wit AI* foi utilizada para o desenvolvimento da aplicação de inteligência, e uma chave de acesso (associada a uma conta) foi gerada para o uso remoto dessa aplicação, sendo que a consulta é feita via "*requests*" enviados aos servidores do *Facebook*. A escolha de usar *Typescript* (MICROSOFT, 2021) em vez do próprio *Javascript* foi apenas o conhecimento prévio dos alunos sobre a linguagem.

A segunda decisão foi em relação a de qual fonte as músicas seriam obtidas. Entre os serviços de música existentes hoje, o mais recomendado, sem custo, é o *YouTube*. Além disso, o *YouTube* apresenta uma API (SEARCH LIST, 2021) de dados de fácil manejo — que, ao se utilizar uma chave de acesso (associada a uma conta), permite fazer consultas de conteúdo.

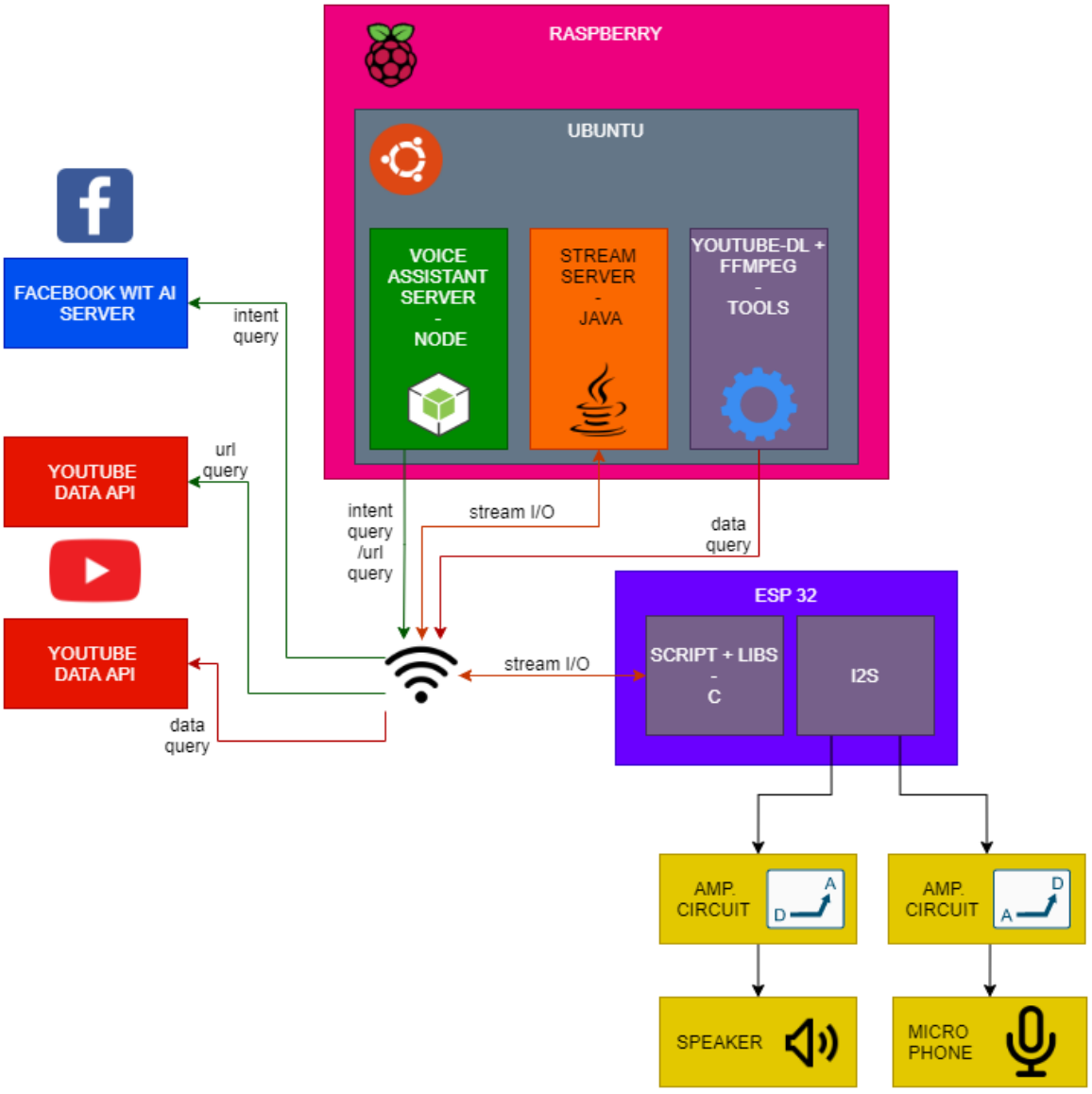
A terceira decisão foi a respeito da forma de obter o áudio dos servidores do *YouTube*. Após pesquisas, foi encontrada a ferramenta chamada *YouTube-DL* (BOLTON, 2020), que permite fazer o download de conteúdo do *YouTube* — e também outra ferramenta, chamada *FFmpeg* (2021), que possibilita converter esse conteúdo (durante o *stream*, o que é muito importante) para o formato desejado.

2 DESENVOLVIMENTO

2.1 Diagrama

Após diversas provas de conceitos, foi escolhido o componente ideal para cada um dos módulos do projeto. Esses componentes estão descritos na seção de desenvolvimento e ilustrados na figura 3 abaixo.

Figura 3 – Diagrama com a descrição da solução do projeto final



Fonte: Autoria própria (2022).

2.1.1 Descrição do diagrama

O bloco rosa representa o *Raspberry Pi 4* (2021), hardware escolhido para ser o servidor da aplicação, atuando como *host* do *Stream Server* (bloco laranja, feito em *Java*) e da assistente de voz (bloco verde, feito em *Typescript* utilizando *Node.js framework*). O sistema operacional escolhido para esse hardware foi o *Raspberry Pi OS*, uma distribuição do *Ubuntu* (CANONICAL, 2021) (bloco cinza). Ainda, representado pelo bloco roxo (interno ao bloco cinza), um conjunto de ferramentas utilizadas pelo servidor, sendo as principais: *FFmpeg* e *YouTube-DL*.

O bloco roxo representa os ESP32 (ESPRESSIF, 2021), hardware escolhido para ser o(s) cliente(s) da aplicação. O ESP32, para seu funcionamento, tem algumas bibliotecas e *drivers* para comunicação *http/socket*, conexão Wi-Fi e utilização do I2S, além do *firmware* em C, desenvolvido pela equipe, que será executado.

Os blocos amarelos representam os circuitos de saída e entrada de áudio, integrados ao ESP32.

Em azul e vermelho (à esquerda), temos o servidor do *Facebook*, onde está hospedada a aplicação *Wit AI*, e o servidor do *YouTube*, utilizado como fonte de *stream* e busca de conteúdo.

2.2 Hardware

Nesta seção, os hardwares utilizados no projeto estão listados junto às suas especificações.

2.2.1 ESP32

Para ser utilizada como cliente, foi escolhido o ESP32, um microcontrolador de baixo custo e baixo consumo que tem Wi-Fi integrado, 36 sinais de GPIO e *Bluetooth* 4.2 (BLE). O quadro 1 contém as especificações do ESP32.

Quadro 1 – Especificações do ESP32

Specifications	ESP32
MCU	Xtensa® Dual-Core 32-bit LX6 600 DMIPS
802.11 b/g/n Wi-Fi	Yes, HT40
Bluetooth	Bluetooth 4.2 and below
Typical Frequency	160 MHz
SRAM	512 kBytes
Flash	SPI Flash , up to 16 MBytes
GPIO	36
Hardware / Software PWM	1 / 16 Channels
SPI / I2C / I2S / UART	4/2/2/2
ADC	12-bit
CAN	1
Ethernet MAC Interface	1
Touch Sensor	Yes
Temperature Sensor	Yes
Working Temperature	-40℃ ~ 125℃

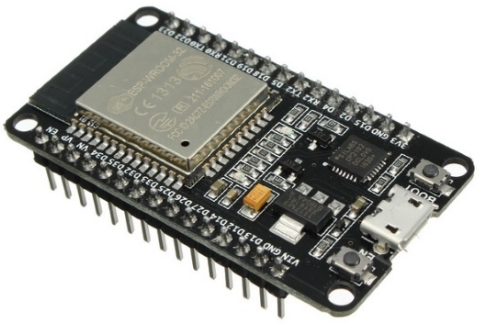
Fonte: Adaptado de CNX SOFTWARE (2021).

Suas características são:

- CPU: Xtensa® Dual-Core 32-bit LX6
- Memória ROM: 448 KBytes
- Clock máximo: 240 MHz
- Memória RAM: 520 KBytes
- Memória Flash: 4 MB
- Wireless padrão 802.11 b/g/n
- Conexão Wi-Fi de 2.4 GHz (máximo de 150 Mbps)
- Antena embutida na placa
- Conector micro USB para comunicação e alimentação
- Wi-Fi Direct (P2P), P2P Discovery, P2P Group Owner mode e P2P Power Management
- Modos de operação: STA/AP/STA+AP
- Bluetooth BLE 4.2
- Portas GPIO: 11
- GPIO com funções de PWM, I2C, SPI e I2S
- Tensão de operação: 4,5 ~ 9 V
- Conversor analógico-digital (ADC)

O modelo está na figura 4 abaixo.

Figura 4 – Modelo do ESP32



Fonte: Google Imagens.

2.2.1.1 I2S

O I2S (*Inter-IC Sound*) é um protocolo de comunicação serial síncrono que é utilizado para fazer a transmissão de áudio entre dois dispositivos digitais. Para o microcontrolador ESP32, existem dois periféricos. Ambos podem ser configurados como saída e como entrada de dados utilizando o I2S *driver*.

O barramento do I2S consiste nas seguintes linhas: *Master clock line*, *Bit clock line*, *Channel select line* e *Serial data line*. Cada controlador I2S tem recursos que podem ser configurados utilizando o I2S *driver*: sistema de operação mestre ou escravo, capacidade de agir como transmissor ou receptor e a utilização de um controlador DMA que permite a transmissão de uma amostra de dados sem precisar usar o CPU para fazer a cópia de cada amostra.

Cada controlador pode operar no modo *half-duplex communication*. Portanto, os dois controladores podem se combinar para estabelecer o modo *full-duplex communication*.

Neste projeto do sistema de som residencial, utilizou-se o protocolo I2S para a saída de áudio com o decodificador UDA1334A.

2.2.1.2 ADC

O conversor analógico-digital é um dispositivo capaz de reproduzir uma representação digital a partir de um processo analógico, normalmente um sinal, por um nível de tensão ou intensidade de corrente elétrica.

Neste projeto, foi utilizado o conversor analógico-digital para a entrada de áudio através do microfone de eletreto MAX9814.

2.2.2 Raspberry Pi 4

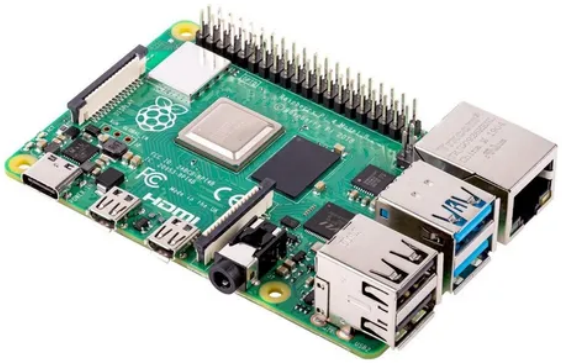
Para ser utilizado como servidor, escolheu-se o *Raspberry Pi 4*, um minicomputador que tem versões com memória RAM de até 8GB DDR4. Para o projeto, foi utilizada a versão de 2GB. No servidor, foi instalado o sistema operacional *Raspberry Pi OS*, que é construído a partir do núcleo *Linux* (2020) e é baseado no *Debian* (2020).

Suas características são:

- Processador: Broadcom (Cortex-A72 quad-core de até 1,5 GHz)
- Memória RAM: 1, 2, 4 ou 8 GB DDR4
- Conectividade: USB-C (energia), USB 3.0, USB 2.0, micro HDMI, microSD, Gigabit Ethernet, Wi-Fi e Bluetooth
- Portas GPIO: 40

O modelo está na figura 5 abaixo.

Figura 5 – Raspberry Pi 4



Fonte: Google Imagens.

2.2.3 Interface com o usuário

Nesta seção, mostra-se, de maneira geral, quais foram os componentes utilizados na ponta final, quando o usuário quer enviar um comando e ouvir os resultados dele, ou então parar de ouvir.

2.2.3.1 Entrada de áudio

O componente utilizado para a entrada de áudio foi o MAX9814. Este componente é um módulo amplificador com microfone eletreto. O modelo está na figura 6 abaixo.

Figura 6 – Microfone eletreto MAX9814

Fonte: Google Imagens.

Suas características são:

- Tensão de alimentação: 2,7 para 5,5 V @ 3 mA
- Saída: 2 Vpp na polarização de 1,25 V
- Resposta de frequência: 20 Hz a 20 KHz
- Ganho máximo: 40 dB, 50 dB ou 60 dB
- Ruído de entrada: 30 nV
- Baixo THD: 0,04% (típico)
- Dimensões: aproximadamente 25 x 15 x 8 mm

2.2.3.2 Saída de áudio

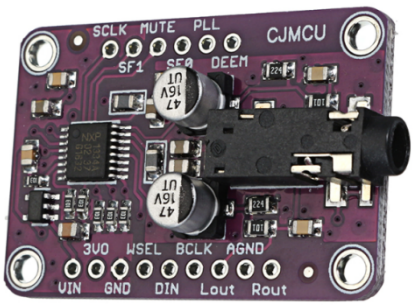
Para a saída de áudio, foi utilizado o módulo amplificador I2S UDA1334A. Este necessita uma baixa tensão para a alimentação, o que ajuda na utilização para este projeto final.

Suas características são:

- Tensão de alimentação: 3,3 para 5 V;
- Filtro digital integrado além do conversor digital-analógico;
- Compatível com protocolo I2S;
- Dimensões: aproximadamente 25 x 38 mm.

O modelo está na figura 7 abaixo.

Figura 7 – UDA1334A



Fonte: Google Imagens.

Na parte dos alto-falantes, adquiriu-se a caixa de som D-02A. Ela contém dois alto-falantes, um potenciômetro para controle de volume e a entrada compatível com *plug* p2.

Suas características são:

- Alto-falantes de dimensão e impedância: 4 Ω;
- Resposta de frequência: 90 Hz-20 KHz;
- Sensibilidade dos alto-falantes: 80 dB;
- SNR: > 80 dB;
- Potência de saída: 5 W;
- Distorção: < 0.3%;
- USB 2.0 & USB 1.1.

O modelo está na figura 8 abaixo.

Figura 8 – Alto-falante com controle de volume D-02A

Fonte: Google Imagens.

2.3 Firmware - ESP32

Nesta seção, apresenta-se o funcionamento do *firmware* em linguagem C, executado no ESP32, onde é feita a leitura e envio dos pacotes de áudio do microfone e a reprodução dos pacotes de áudio recebidos.

No projeto definido, é necessário o uso de *tasks* para serem executadas em paralelo (pseudoparalelismo), garantindo que a entrada e saída de áudio sejam administradas simultaneamente, sem interrupções.

Para alcançar isso, foram utilizadas as funções genéricas do sistema operacional do ESP32, o *FreeRTOS* (ESPRESSIF, 2021).

O principal recurso foi o método de criação de uma *task*, na figura 9 abaixo.

Figura 9 – Trecho de código do firmware - setupKeepAlive

```
void setupKeepAlive() {
  xTaskCreatePinnedToCore(keepAliveTask, "Keep Alive Task", 2048, NULL, 1, NULL, 1);
}
```

Fonte: Autoria própria (2022).

Os parâmetros que configuram essa função são, respectivamente:

- O método que a *task* irá executar;

- Nome dado à *task*;
- O tamanho da pilha alocada à *task*;
- Ponteiro do parâmetro a ser passado como argumento ao método da *task*;
- Prioridade que a *task* tem para ser escalonada;
- Identificador da *task*;
- ID do processador, caso seja definido que apenas uma CPU específica vá escalonar essa *task*.

Além do *FreeRTOS*, também foram utilizadas as funções dos *drivers* de I2S e ADC, instalados no ESP32, permitindo o uso de métodos já implementados, tanto para a escrita de *bytes* no *buffer* de saída (saída de áudio), quanto na leitura dos *bytes* no *buffer* de entrada (entrada de áudio). Também foi importado o *driver* de conexão Wi-Fi. Alguns métodos utilizados podem ser vistos nas figuras 10, 11 e 12 abaixo.

Figura 10 – Trecho de código do firmware - i2s_read

```
// read from i2s
i2s_read(sampler->getI2SPort(), i2sData, 1024, &bytesRead, 10);
```

Fonte: Autoria própria (2022).

Figura 11 – Trecho de código do firmware - i2s_write

```
// write on i2s
i2s_write(i2s_num, samples + bufferIdx, NUM_BYTES_TO_READ_FROM_FILE - bufferIdx, &bytesWritten, 1);
```

Fonte: Autoria própria (2022).

Figura 12 – Trecho de código do firmware - Configuração WiFi

```
WiFi.mode(WIFI_STA);
WiFi.begin(SSID, PASSWORD);
if (WiFi.waitForConnectResult() != WL_CONNECTED)
{
  Serial.println("Connection to Wi-Fi failed! Rebooting...");
  delay(1000);
  ESP.restart();
}
```

Fonte: Autoria própria (2022).

Ainda, utilizou-se como base do algoritmo de leitura de dados do ADC (e modificado para esse sistema) o código de *loop_sampling* do projeto *atomic14/esp32_audio* (CHRIS, 2021). Esse projeto permite criar um objeto do tipo *ADCSampler*, que faz o controle do *buffer* de entrada e é responsável por notificar a *task* que envia esses dados para o *Stream Server*. Na figura 13 abaixo, é demonstrado o método do objeto *ADCSampler*, responsável por esse controle.

Figura 13 – Trecho de código do firmware - i2sReaderTask

```
void i2sReaderTask(void *param)
{
    I2SSampler *sampler = (I2SSampler *)param;
    while (true)
    {
        // wait for some data to arrive on the queue
        i2s_event_t evt;
        if (xQueueReceive(sampler->m_i2sQueue, &evt, portMAX_DELAY) == pdPASS)
        {
            if (evt.type == I2S_EVENT_RX_DONE)
            {
                size_t bytesRead = 0;
                do
                {
                    // read data from the I2S peripheral
                    uint8_t i2sData[1024];
                    // read from i2s
                    i2s_read(sampler->getI2SPort(), i2sData, 1024, &bytesRead, 10);
                    // process the raw data
                    sampler->processI2SData(i2sData, bytesRead);
                } while (bytesRead > 0);
            }
        } else {
          yield();
        }
    }
}
```

Fonte: Autoria própria (2022).

2.3.1 ADC Input

Trata-se de como são transformados os sinais obtidos pelo microfone para serem enviados ao servidor e então gravados em arquivos de áudio. Esses sinais de voz chegam ao microcontrolador como sinais digitais provenientes de uma conversão do sinal analógico (voz) obtido pelo microfone.

2.3.1.1 Prova de conceito

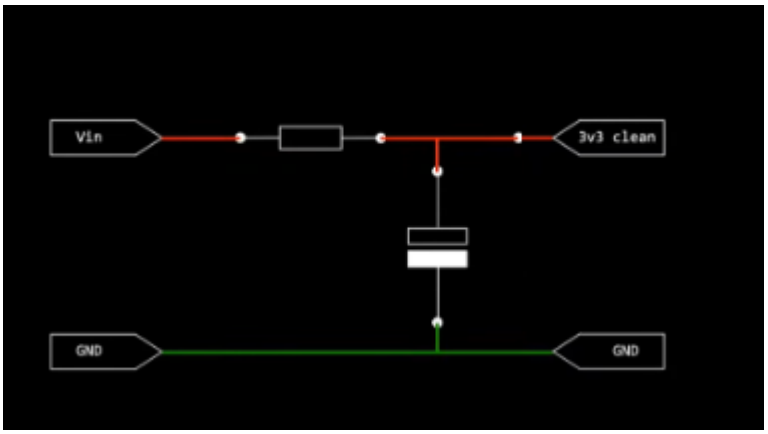
Para a prova de conceito da entrada de áudio, primeiro foram feitas pesquisas sobre os microfones, que têm algumas características: são compatíveis com as entradas que utilizam o conversor analógico-digital (ADC) — ou com as entradas I2S do microcontrolador ESP32, os quais têm boa qualidade de áudio. A ideia para esse sistema foi registrar comandos de voz através do microfone conectado ao ESP32.

Durante as pesquisas, notou-se que os microfones compatíveis com a entrada I2S do microcontrolador ESP32 têm melhor qualidade e menor ruído se comparados aos microfones compatíveis com a entrada ADC.

Utilizando a alimentação de 3v3 do ESP32, existe um ruído adicionado ao áudio de entrada registrado, porém é possível reduzi-lo utilizando um filtro Passa-Baixa na entrada de alimentação do módulo amplificador com microfone de eletreto.

O esquemático do filtro na entrada está exemplificado na figura 14 abaixo.

Figura 14 – Filtro Passa-Baixa na entrada do microfone



Fonte: Adaptado de CHRIS (2021).

Para o valor do resistor, é utilizado 22 ohms e, para o valor do capacitor, é utilizado 1000 uF. A conexão Vin é conectada ao sinal 3v3 do ESP32 e a conexão 3v3 clean é conectada à porta Vdd do microfone MAX9814. Já cada conexão GND é conectada ao GND do ESP32 e também do microfone.

Existem algumas maneiras para a utilização de um áudio analógico na entrada do ESP32. Uma dessas formas é fazer a leitura utilizando o conversor

analógico-digital (ADC) do microcontrolador ESP32. Porém essa forma apresenta limitações, pois é utilizada para leituras únicas, não para leituras com altas taxas de amostragem.

A ESP32 possui a opção de utilizar o buffer DMA (Direct Memory Access) destinado ao protocolo I2S juntamente com o ADC. Dessa forma, podemos ter melhor performance com maior taxa de amostragem. Esta forma é compatível com os microfones eletreto MAX4466 e MAX9814. Este foi o método utilizado.

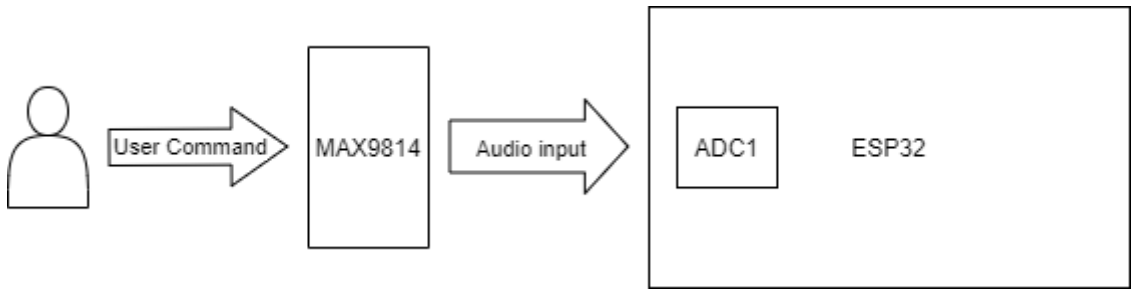
Também existe a possibilidade de fazer a leitura do áudio diretamente da entrada I2S da ESP32, porém, para utilizar essa forma, são necessários periféricos com suporte para I2S, como os microfones SPH0645LM4H, INPM441, ICS43432 e ICS43434.

Para fazer a leitura diretamente do ADC do ESP32, existem duas entradas: ADC1 e ADC2. O ADC1 contém 8 canais: GPIO32 até GPIO39. Já o ADC2 contém 10 canais: GPIO4, GPIO0, GPIO2, GPIO15, GPIO13, GPIO12, GPIO14, GPIO27, GPIO25 e GPIO26.

2.3.1.2 Implementação

Nos primeiros testes feitos para a entrada de áudio, foi utilizado o microfone eletreto MAX9814 e a entrada ADC1 com GPIO35. A figura 15 abaixo mostra a relação entre o comando de voz do usuário, o sensor de áudio e o ESP32.

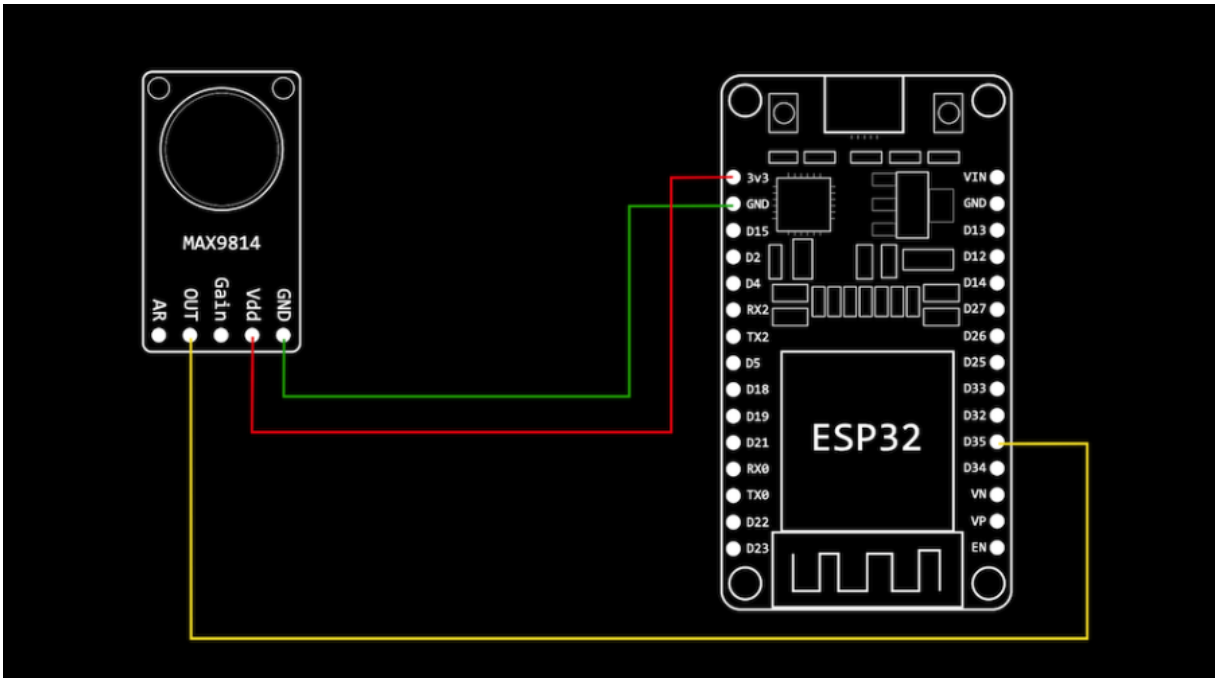
Figura 15 – Diagrama entre comando de voz, sensor de áudio e ESP32



Fonte: Autoria própria (2022).

As entradas conectadas ao ESP32 estão representadas na figura 16 abaixo.

Figura 16 – Microfone eletreto conectado ao ESP32



Fonte: Adaptado de CHRIS (2021).

A implementação do algoritmo de gravação, tanto sua inicialização até o envio os dados, é dada pelas seguintes funções:

setupMicrophone(): essa função é chamada apenas quando a placa é inicializada. O objetivo dela é inicializar a configuração I2S para o microfone, inicializar o objeto *ADCSampler* da biblioteca importada e criar uma *task* que deve constantemente formatar e enviar os dados ao servidor — essa *task* executa o método **adcWriterTask()**. Representada na figura 17.

Figura 17 – Trecho de código do firmware - setupMicrophone

```
void setupMicrophone() {
  wifiClientADC = new WiFiClient();
  httpClientADC = new HTTPClient();
  httpClientADC->begin(*wifiClientADC, ADC_SERVER_URL);
  httpClientADC->addHeader("content-type", "application/octet-stream");
  adcSampler = new ADCSampler(ADC_UNIT_1, ADC1_CHANNEL_7);
  TaskHandle_t adcWriterTaskHandle;
  xTaskCreatePinnedToCore(adcWriterTask, "ADC Writer Task", 4096, adcSampler, 1, &adcWriterTaskHandle, 1);
  adcSampler->start(adc_num, adcI2SConfig, 2048, adcWriterTaskHandle, ROOM);
}
```

Fonte: Autoria própria (2022).

adcWriterTask(void *param): a *task* monitora um semáforo, que é liberado quando o *buffer* de dados obtidos do microfone chega ao tamanho do pacote

definido para ser enviado. Esse pacote é enviado pelo método **sendData()**. Representada na figura 18.

Figura 18 – Trecho de código do firmware - adcWriterTask

```
void adcWriterTask(void *param)
{
  I2SSampler *sampler = (I2SSampler *)param;
  const TickType_t xMaxBlockTime = pdMS_TO_TICKS(100);
  while (true) {
    // wait for some samples to save
    uint32_t ulNotificationValue = ulTaskNotifyTake(pdTRUE, xMaxBlockTime);
    if (ulNotificationValue > 0) {
      sendData(httpClientADC, (uint8_t *)sampler->getCapturedAudioBuffer(), sampler->getBufferSizeInBytes(), sampler->getRoom());
    } else {
      yield();
    }
  }
}
```

Fonte: Autoria própria (2022).

O semáforo é controlado pelo objeto *ADCSampler*, mencionado anteriormente.

sendData(HTTPClient *httpClient, uint8_t *bytes, size_t count, String room): por fim, cada bloco de dados recebido e tratado é enviado para o *server*, que deve receber os blocos e adicioná-los a um arquivo de áudio. É utilizado o protocolo HTTP para enviar um *POST Request* ao servidor, contendo o cômodo (room*)* no *header* e os *bytes* do bloco no *body*. Representado na figura 19.

Figura 19 – Trecho de código do firmware - sendData

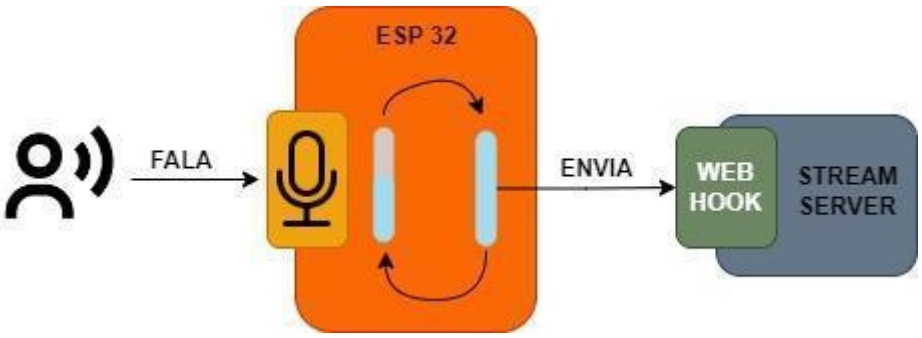
```
void sendData(HTTPClient *httpClient, uint8_t *bytes, size_t count, String room)
{
  httpClient->addHeader("room-name", room);
  httpClient->POST(bytes, count);
}
```

Fonte: Autoria própria (2022).

Para garantir um funcionamento sem interrupção, a cada sinalização, é trocado o buffer usado para escrita do áudio gravado. Desta forma, é possível continuar gravando enquanto o áudio está sendo processado para envio.

A figura 20 abaixo ilustra o algoritmo de gravação de áudio na ESP32.

Figura 20 – Diagrama de funcionamento da entrada de áudio na ESP32



Fonte: Autoria própria (2022).

2.3.2 I2S Output

Algoritmo utilizado para transformar o arquivo de áudio buscado na biblioteca do *YouTube* e reproduzido nos alto-falantes. O arquivo de áudio é recebido pelo ESP32 em blocos. O áudio é formatado com a ferramenta *FFmpeg* para um formato digital, porém sem compressão e enviado para a saída de áudio, utilizando o protocolo I2S para essa comunicação entre ESP32 e UDA1334A.

2.3.2.1 Prova de conceito

Para a prova de conceito da saída de áudio, primeiro foram feitas pesquisas sobre os possíveis componentes que são compatíveis com o microcontrolador ESP32.

Entre as possibilidades, existem o Conversor Digital Analógico (DAC) e o I2S. Para os primeiros testes, foi utilizado o DAC do ESP32.

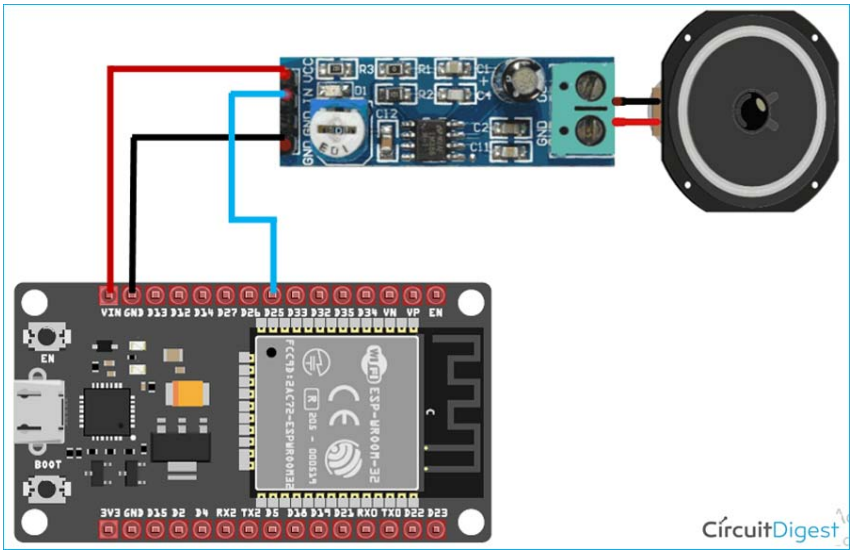
Nas pesquisas feitas para a utilização do DAC, o módulo LM386 foi encontrado como uma solução simples e de fácil integração com o sistema. Porém, para obter um áudio com qualidade superior, foi utilizado o decodificador de áudio UDA1334A.

2.3.2.1.1 LM386

Os componentes utilizados para o primeiro projeto da saída de áudio foram: microcontrolador ESP32 *Doit Devkit V1*, módulo amplificador LM386, alto-falante de 4 ohms com 3W e *jumpers* para conexão.

O diagrama do circuito está na figura 21 abaixo.

Figura 21 – Diagrama do circuito utilizando o módulo amplificador LM386

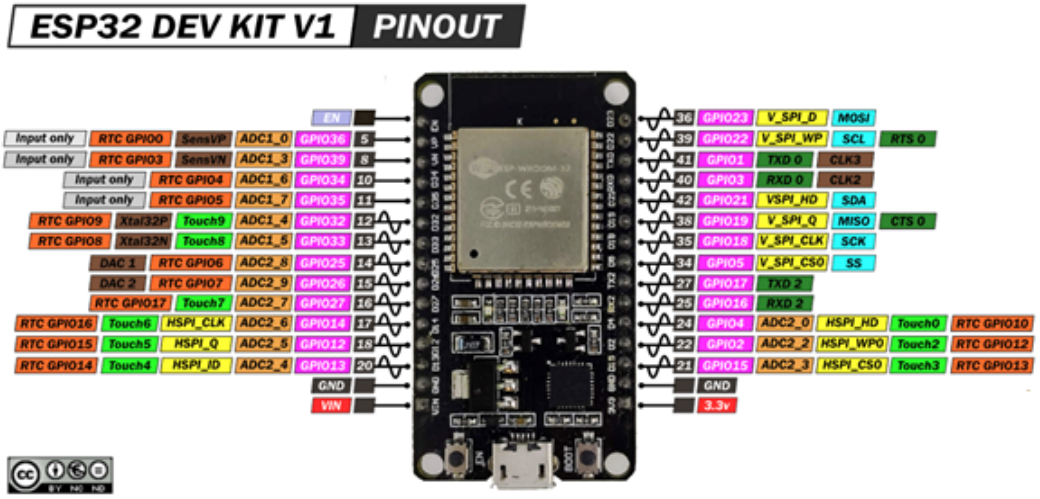


Fonte: Adaptado de CNX SOFTWARE (2021).

Para as conexões do módulo amplificador ao ESP32, os pinos GND e VCC do módulo LM386 foram conectados aos pinos GND e VIN do ESP32. O pino IN do módulo amplificador foi conectado ao pino GPIO 25, que deve ser configurado para a utilização da I2S do ESP32.

O GPIO 25 é um dos pinos que pode ser configurado como DAC no ESP32. O outro pino existente é o GPIO 26. Os pinos do ESP32 estão ilustrados na figura 22 abaixo.

Figura 22 – Diagrama dos pinos existentes no ESP32



Fonte: Adaptado de ELECTRORULES (2020).

Para a prova de conceito, foi testado com taxa de amostragem de 8 kHz e 16 kHz. Também é necessário utilizar o tipo de codificação “Unsigned 8-bit PCM”.

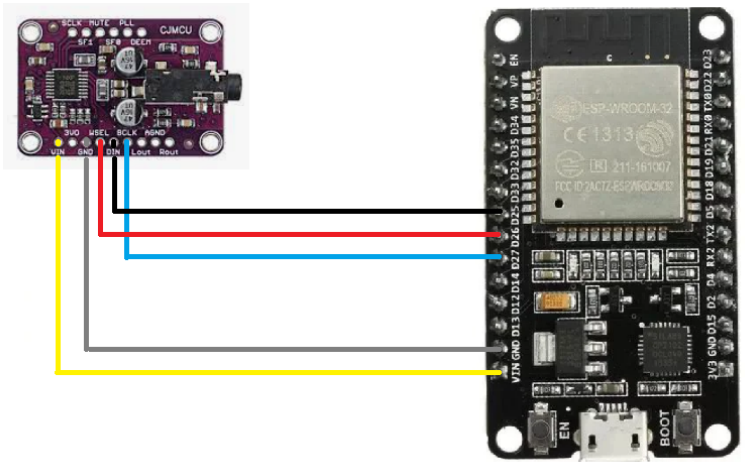
Inicialmente, para fins de testes, anterior a integração do Stream Server, foi utilizado a ferramenta Audacity (2020) para gerar amostras de áudio no formato citado acima. Para armazenamento das amostras de áudio no cliente (ESP32), o áudio foi convertido para hexadecimal. Para isso, utilizou-se a ferramenta Hex Editor.

2.3.2.1.2 UDA1334A

Para o projeto final da saída de áudio, foram utilizados: microcontrolador ESP32 *Doit Devkit V1*, decodificador de áudio UDA1334A, dois alto-falantes de 4 ohms com 3W e *jumpers* para conexão.

Para as conexões do decodificador de áudio com o ESP32, os pinos GND e VCC do decodificador foram conectados aos pinos GND e VIN do ESP32. Os pinos de *Bit Clock Connection* (BCLK), *Word Select* (WSEL) e *Data In* (DIN) foram conectados aos pinos 27, 26 e 25, respectivamente. As conexões estão na figura 23.

Figura 23 – Diagrama do projeto com o decodificador UDA1334A



Fonte: Adaptado de Google Imagens.

2.3.2.2 Implementação

A implementação da saída de áudio foi uma das partes mais desafiadoras do projeto. No código final funcional, foi utilizado o seguinte algoritmo no ESP32:

Duas *tasks* ativas: uma responsável pela leitura dos dados (obtidos via conexão *socket* com o *stream server*) e outra responsável pelo monitoramento e escrita no *buffer* (via DMA) do I2S, que é lido/reproduzido automaticamente. Estão na figura 24.

Em primeiro lugar, criam-se algumas constantes de configuração, como taxas de transferência, portas utilizadas, entre outras. A maioria desses valores foram obtidos de referências, outros a partir de experimentos locais.

Figura 24 – Trecho de código do firmware - Configuração i2s

```
// ---- I2S CONFIGURATION ----
static const i2s_config_t i2s_config =
{
  .mode = (i2s_mode_t)(I2S_MODE_MASTER | I2S_MODE_TX),
  .sample_rate = SAMPLE_RATE,
  .bits_per_sample = I2S_BITS_PER_SAMPLE_16BIT,
  .channel_format = I2S_CHANNEL_FMT_RIGHT_LEFT,
  .communication_format = (i2s_comm_format_t)(I2S_COMM_FORMAT_I2S | I2S_COMM_FORMAT_I2S_MSB),
  .intr_alloc_flags = ESP_INTR_FLAG_LEVEL1,
  .dma_buf_count = 8,
  .dma_buf_len = 1024,
  .use_apll = true,
  .tx_desc_auto_clear = true,
  .fixed_mclk = -1
};

static const i2s_pin_config_t pin_config =
{
  .bck_io_num = 27,                         // The bit clock connectiom, goes to pin 27 of ESP32
  .ws_io_num = 26,                          // Word select, also known as word select or left right clock
  .data_out_num = 25,                       // Data out from the ESP32, connect to DIN on 38357A
  .data_in_num = I2S_PIN_NO_CHANGE          // we are not interested in I2S data into the ESP32
};
```

Fonte: Autoria própria (2022).

A função **setupSpeaker()** é chamada apenas no momento da inicialização. Ela é responsável por criar e iniciar a *task* “Speaker Task”, referente ao recebimento dos dados, além de inicializar os pinos e o *driver* para utilizar o I2S.

Figura 25 – Trecho de código do firmware - setupSpeaker

```
void setupSpeaker() {
  i2s_driver_install(i2s_num, &i2s_config, 0, NULL);
  i2s_set_pin(i2s_num, &pin_config);
  i2s_set_sample_rates(i2s_num, SAMPLE_RATE);
  xTaskCreatePinnedToCore(speakerTask, "Speaker Task", 8192, &dataBuffer, 1, NULL, 1);
}
```

Fonte: Autoria própria (2022).

As funções referentes à saída de áudio rodam em duas partes principalmente: receber e reproduzir os dados do servidor.

A primeira parte é responsabilidade da *task* de recebimento, que executa o método **getData()** em *loop*.

Esse método envia uma requisição ao servidor, notificando um pedido de 1024 *bytes* via *socket*. A *task* armazena os dados em um *buffer* e altera a *flag* **bufferFull**, sinalizando que o mesmo está completo. Nas próximas execuções, antes

de ler os dados e armazená-los, o estado dessa *flag* é verificado e precisa indicar um *buffer* vazio para a execução prosseguir. A *task* não aguarda o semáforo para notificar o servidor, apenas para ler os dados. As funções estão representadas na figura 26.

Figura 26 – Trecho de código do firmware - speakerTask e getData

```
void speakerTask(void *param) {
  while (true) {
    getData((uint8_t*) param);
  }
}

void getData(uint8_t* arr) {
  client.write( B11111111 );
  while (client.available() == 0) {
    yield();
  }
  while (bufferFull) {
    yield();
  }
  for (int i = 0; i < NUM_BYTES_TO_READ_FROM_FILE; i++) {
    arr[i] = client.read();
  }
  bufferFull = true;
}
```

Fonte: Autoria própria (2022).

A segunda parte é responsabilidade do *loop* principal do código, que executa constantemente a função **playWav()**, conforme apresentado na figura 27.

Figura 27 – Trecho de código do firmware - playWav, fillI2SBuffer

```
void playWav()
{
  static bool readingFile = true;
  static byte samples[NUM_BYTES_TO_READ_FROM_FILE];
  static uint16_t bytesRead;

  if (readingFile) {
    readData(samples);
    readingFile = false;
  } else
    readingFile = fillI2SBuffer(samples);
}

void readData(byte* samples) {
  while (!bufferFull) {
      yield();
  }
  memcpy(samples, dataBuffer, NUM_BYTES_TO_READ_FROM_FILE);
  bufferFull = false;
}

bool fillI2SBuffer(byte* samples)
{
  size_t bytesWritten;
  static uint16_t bufferIdx = 0;

  i2s_write(i2s_num, samples + bufferIdx, NUM_BYTES_TO_READ_FROM_FILE - bufferIdx, &bytesWritten, 1);
  bufferIdx += bytesWritten;

  if (bufferIdx >= NUM_BYTES_TO_READ_FROM_FILE) {
    bufferIdx = 0;
    return true;
  } else
    return false;
}
```

Fonte: Autoria própria (2022).

Este método tem dois modos de funcionamento, sinalizados pela *flag* **readingFile**.

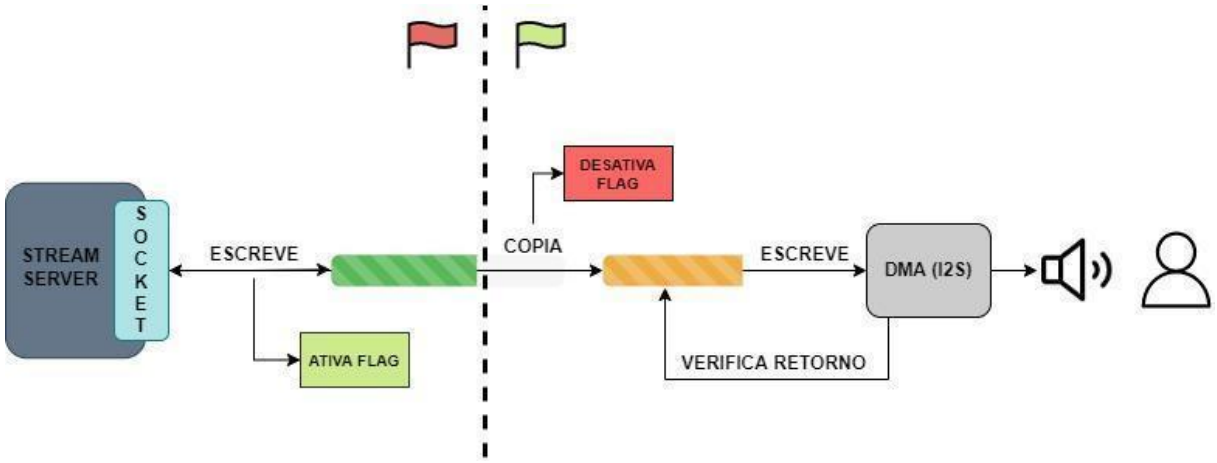
No primeiro momento, esta *flag* está ativa, indicando que deve-se executar a operação de leitura. Esta operação move os dados do *buffer* de dados para um *buffer* auxiliar e sinaliza para a *task* de recebimento que o *buffer* de dados está disponível (desabilita a *flag* **bufferFull**), permitindo que a leitura e armazenamento de dados prossiga. Essa operação também desativa a *flag* **readingFile**, ativando o segundo modo de funcionamento.

Neste modo, os dados são continuamente escritos no *buffer* DMA do I2S, para serem automaticamente reproduzidos.

Em cada escrita, o número de bytes de fato escritos no DMA pode não ser o número de bytes passado no método. A verificação deste valor garante que não aconteça sobreposição de dados. Esse processo ocorre até que todos os bytes do buffer auxiliar sejam corretamente recebidos e reproduzidos. Ao final, a *flag* **readingFile** é ativa novamente, trocando novamente o modo de operação.

Dessa forma, conseguimos criar um fluxo de dados ininterrupto em velocidade aceitável para a reprodução do áudio. A figura 28 abaixo ilustra o algoritmo de reprodução de áudio na ESP32.

Figura 28 – Diagrama de funcionamento da saída de áudio na ESP32



Fonte: Autoria própria (2022).

2.3.3 Suporte para múltiplos clientes

Para a sincronização de comunicações simultâneas, no momento de inicialização do cliente, o programa envia uma requisição de identificação ao servidor, informando o cômodo configurado.

Para isso, interno ao ESP32, criou-se um arquivo de configuração *hard coded* chamado *Room.h*, que define o nome do cômodo em questão. O arquivo de configuração e o método de identificação podem ser vistos abaixo, nas figuras 29 e 30.

Figura 29 – Arquivos de configuração do ESP32. Em detalhe, o Room.h

Fonte: Autoria própria (2022).

Figura 30 – Trecho de código do firmware - setupIdentity

```
void setupIdentity() {
  WiFiClient wClient;
  HTTPClient hClient;
  hClient.begin(wClient, CONNECTION_SERVER_URL);
  hClient.addHeader("Content-Type", "application/json");
  String body = (String) "{\"room\":\"" + ROOM + "\",\"ip\":\"" + WiFi.localIP().toString().c_str() + "\"}";
  if(hClient.POST(body) != 200) {
    Serial.println("Failed to set ID for this room on Stream Server. Restarting ...");
    ESP.restart();
  }
  hClient.end();
}
```

Fonte: Autoria própria (2022).

Como observado no método, uma requisição POST é enviada do cliente para o servidor com o *payload* em formato JSON contendo a informação do cômodo.

2.3.4 Watchdog

Para o controle de conexão entre o servidor e o cliente, utilizou-se um sistema de *watchdog*. Foi criado do lado do *Stream Server* um *endpoint REST* do tipo GET para receber esse controle.

Na parte do cliente, foi criada uma *task* periódica para mandar uma requisição HTTP do tipo GET para esse *endpoint GET /keepalive*. Caso a resposta dessa requisição não seja positiva (diferente de 200 OK), a *task* gera um *reboot* do ESP32, para que a conexão entre o cliente e o servidor seja restabelecida. A *task* é executada a cada um segundo.

A figura 31 abaixo mostra a função que é executada periodicamente.

Figura 31 – Trecho de código do firmware - sendKeepAlive

```
void sendKeepAlive() {
  WiFiClient wClient;
  HTTPClient hClient;
  hClient.begin(wClient, KEEPALIVE_SERVER_URL);
  hClient.addHeader("Content-Type", "application/json");
  if(hClient.GET() != 200) {
    Serial.println("Failed to connect to server. Restarting client ...");
    ESP.restart();
  }
  hClient.end();
}
```

Fonte: Autoria própria (2022).

A figura 32 e 33 abaixo mostra o método sendo chamado, em *loop*.

Figura 32 – Trecho de código do firmware - keepAliveTask

```
void keepAliveTask(void *param) {
  while(true) {
    sendKeepAlive();
    delay(1000);
  }
}
```

Fonte: Autoria própria (2022).

Figura 33 – Trecho de código do firmware - setup

```
void setup()
{
  Serial.begin(115200);
  setupNetwork();
  setupMicrophone();
  setupSpeaker();
  setupIdentity();
  setupKeepAlive();
  Serial.println("ESP32 Setup completed!");
}
```

Fonte: Autoria própria (2022).

2.3.5 Raspberry Pi OS

A preparação do ambiente no servidor implementado na *Raspberry Pi* na placa de desenvolvimento se dá em seis principais passos: *Python* 3.9, *FFmpeg*, *YouTube DL*, *Node* v10+, *Java* 8+ (ORACLE, 2021) e *Maven* (APACHE, 2020).

Java

Para a instalação do Java, foi usada esta referência:
Como instalar o Java com o Apt no Ubuntu 20.04 | DigitalOcean

Figura 34 – Comandos de instalação do Java JRE/JDK

```
sudo apt update
sudo apt install default-jre
sudo apt install default-jdk
```

Fonte: Autoria própria (2022).

Maven

A instalação do *Maven* foi necessária para a compilação do *Stream Server*. Além de descompactar, foi necessário adicionar o diretório do *maven* às variáveis de sistema, seguindo as seguintes referências:

Installing Maven on Raspberry Pi 3 OS Raspbian - Stack Overflow
Installing maven on the Raspberry Pi – Xianic Blog

Figura 35 – Comandos de instalação do Maven

```
wget https://www-eu.apache.org/dist/maven/maven-3/3.6.1/binaries/apache-maven -3.6.1-bin.tar.gz

cd /opt && sudo tar -xzvf /home/pi/Downloads/apache-maven-3.6.1-bin.tar.gz

vi ~/.bashrc -> Add the following lines and save the file.
    export M2_HOME=/opt/apache-maven-3.2.5
    export "PATH=$PATH:$M2_HOME/bin"

source ~/.bashrc
```

Fonte: Autoria própria (2022).

Node

O Node.js é necessário para compilar e executar a Assistente de Voz. Os passos para a instalação do *Node*:

Figura 36 – Comandos de instalação do Node.js

```
sudo apt install nodejs
sudo apt install npm
```

Fonte: Autoria própria (2022).

Python 3.9

O Python é uma dependência necessária para execução do Youtube-DL em sistemas Linux. Para a instalação do *python*, foram seguidos passos majoritariamente desta referência:

Install Python 3.9.5 on Raspberry Pi step by step

Em resumo, deve-se baixar, extrair e instalar:

Figura 37 – Comandos de download do Python 3.9.5

```
wget https://www.python.org/ftp/python/3.9.5/Python-3.9.5.tar.xz
tar -Jxf Python-3.9.5.tar.xz
cd Python-3.9.5
./configure --enable-optimizations --prefix=/usr
make
sudo make install
```

Fonte: Autoria própria (2022).

Por último, é necessário trocar o *alias* de execução do *python* para o executável da versão 3.9, caso contrário, o *python* 2.7 será usado:

Figura 38 – Comandos de instalação do Python 3.9.5

```
echo "alias python='/usr/local/bin/python3.9'" >> ~/.bashrc
. ~/.bashrc
```

Fonte: Autoria própria (2022).

FFmpeg

Para a instalação do *FFmpeg*, foram seguidos os passos do guia abaixo:

How to Install and Use FFmpeg on Ubuntu 20.04 | Linuxize

Figura 39 – Comandos de instalação do FFmpeg

```
sudo apt install ffmpeg
```

Fonte: Autoria própria (2022).

Youtube DL

Foram usadas duas referências quanto ao *YouTube DL*. A primeira, majoritariamente:

youtube-dl/README.md at master

Install youtube-dl on Raspberry Pi | Lindevs

Foi baixado o *Youtube DL*, alterada a permissão do arquivo executável e movido para o diretório final do projeto.

Figura 40 – Comandos de instalação do Youtube-DL

```
sudo wget https://yt-dl.org/downloads/latest/youtube-dl -O /usr/local/bin/youtube-dl
sudo chmod a+rx /usr/local/bin/youtube-dl

mv youtube-dl /opt/application/tools/
```

Fonte: Autoria própria (2022).

2.3.6 Raspberry Pi – Stream Server + Ai

Depois de ter todos os softwares necessários instalados na *Raspberry Pi*, é necessário iniciar o servidor.

Para efeitos facilitadores, antes de tudo, troca-se o usuário para *root*. As *tools FFmpeg* e *YouTube DL* foram movidas para um diretório de produção.

Figura 41 – Comandos de configuração do sistema

```
sudo su



cd /usr/local/bin/youtube-dl
mv youtube-dl /opt/application/tools/
cd /opt/application/tools/
./youtube-dl --version

cd /usr/bin/ffmpeg
mv ffmpeg /opt/application/tools/
cd /opt/application/tools/
./ffmpeg -h
```

Fonte: Autoria própria (2022).

Feito o download do código no *GitHub* (SOUZA, 2020), o projeto é compilado e os arquivos finais movidos ao diretório de produção, de onde executa-se as aplicações.

Figura 42 – Comandos de inicialização do *Stream Server*

```
cd /home/pi/Documents/tcc-full/tcc/wifi/server-java/
mvn clean install
cd target/
cp /home/pi/Documents/tcc-full/tcc/wifi/server-java/target/server-java-0.0.1-SNAPSHOT.jar /opt/application/stream-server/
cd /opt/application/stream-server/
java -jar ./server-java-0.0.1-SNAPSHOT.jar
```

Fonte: Autoria própria (2022).

Para a execução da assistente de voz:

Figura 43 – Comandos de configuração da Assistente de Voz

```
cd /home/pi/Documents/tcc-full/tcc/voice-assistant/backend
npm run start
```

Fonte: Autoria própria (2022).

2.4 Software

O pacote de software, executado no servidor, pode ser dividido em duas partes.

A primeira, a Assistente de voz é responsável pela comunicação com o servidor de inteligência artificial *Wit AI* e com os serviços do *YouTube*.

A segunda, o *Stream Server* é responsável por receber, tratar e enviar pacotes de áudio para o ESP32.

2.4.1 Assistente de voz

A assistente de voz é uma aplicação *Node.js*, implementada em *Typescript*, que se comunica com o *Stream Server* com o intuito de obter as gravações a serem enviadas à *Wit AI* e processar os resultados e tomadas de decisões.

A aplicação é também responsável por encontrar, através da API do *YouTube*, a música escolhida e iniciar o download do áudio.

2.4.1.1 Prova de conceito

Primeiro projeto

Primeiramente, a assistente de voz foi projetada para operar em um ambiente de desenvolvimento *Windows*, com acesso a dispositivos de entrada e saída de áudio locais.

Para isso, o design da aplicação era composto por um serviço principal e três serviços adjuntos, da seguinte forma:

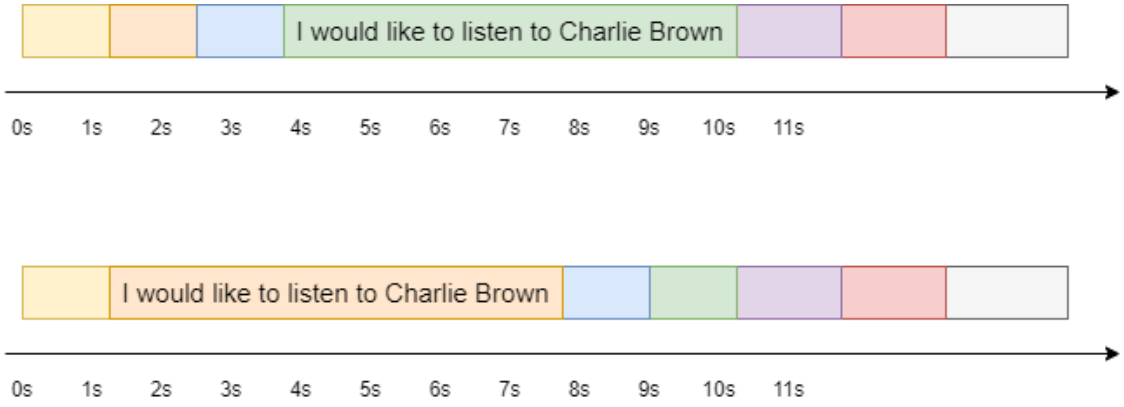
- Serviço principal: responsável pela execução e delegação de comandos aos serviços adjuntos.

- Serviço de áudio: responsável pelo acesso e uso dos dispositivos de entrada e saída de áudio.
- Serviço de acesso ao *YouTube*: responsável pela consulta na API de dados do *YouTube* e download do conteúdo via ferramentas do projeto.
- Serviço de acesso à *Wit AI*: responsável por enviar os trechos de áudio gravados e analisar o comando de voz.

A execução, resumidamente, ocorre da seguinte forma:

O serviço de áudio inicia gravações de seis segundos (valor aproximado de um comando de voz) a cada um segundo, gerando assim "*frames*" de seis segundos de gravação que são enviados a cada segundo para o aplicativo da *Wit AI* (via serviço da *Wit AI*). Isso remove a necessidade de algum botão ou outra forma de iniciar um comando, delegando à inteligência artificial o trabalho de decidir se de fato foi um comando recebido. A figura 44 abaixo mostra o comando.

Figura 44 – Representação do comando de voz



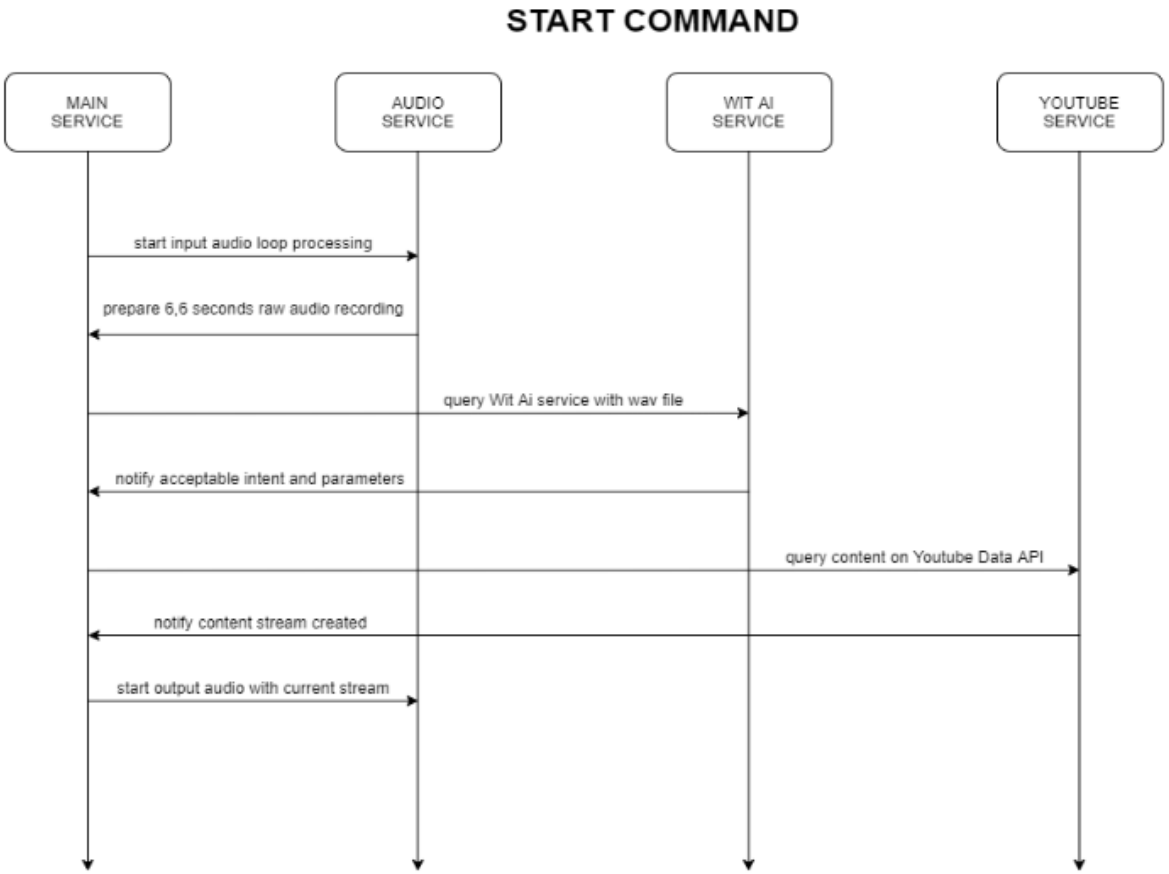
Fonte: Autoria própria (2022).

Quando o serviço da *Wit AI* decide (existem critérios de decisão baseados nas estatísticas recebidas) que a resposta recebida corresponde a um pedido de "*play*" ou "*stop*", é delegado ao serviço de acesso ao *YouTube* um "*request*" para a busca do conteúdo com base na "*keyword*" do comando.

Quando o *stream* (entrante) do conteúdo da música é iniciado, é delegada ao serviço de áudio a reprodução no dispositivo definido.

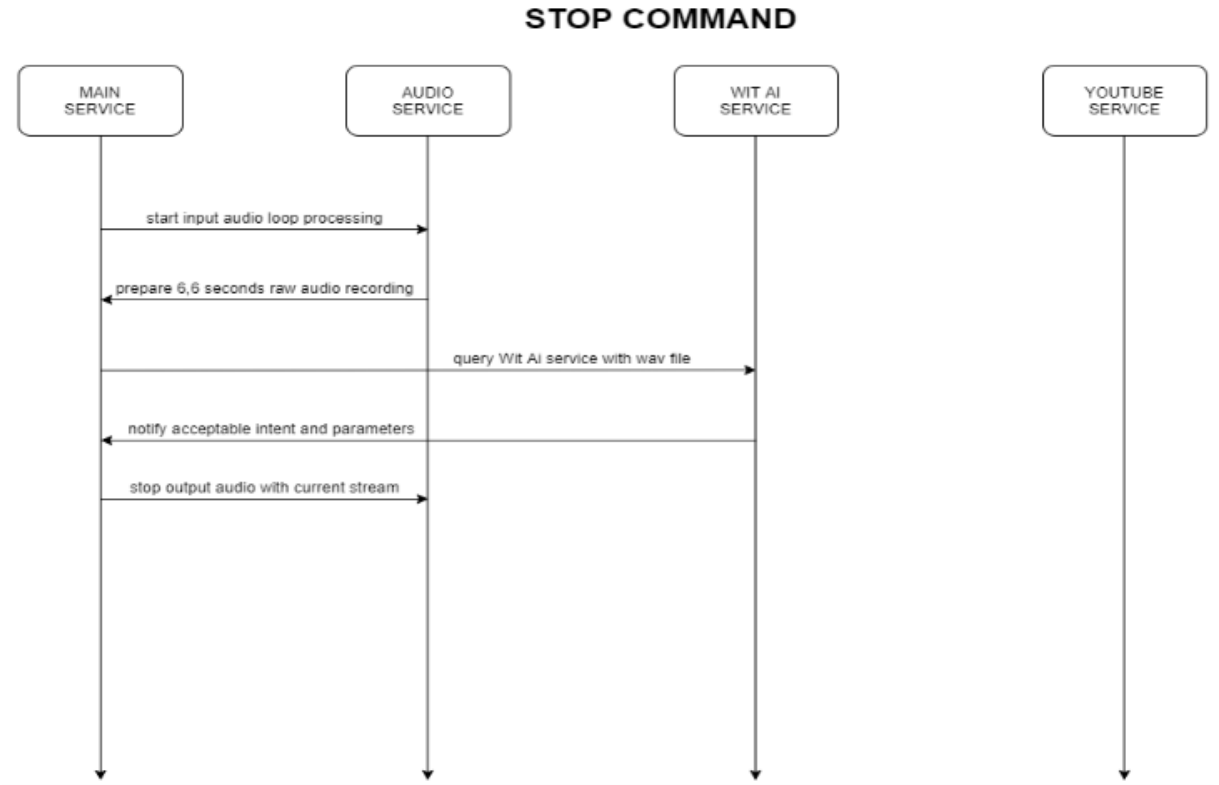
A execução desses comandos podem ser vistos nas figuras 45 e 46 abaixo.

Figura 45 – Diagrama de tempo para execução do comando *play*



Fonte: Autoria própria (2022).

Figura 46 – Diagrama de tempo com execução do comando *stop*



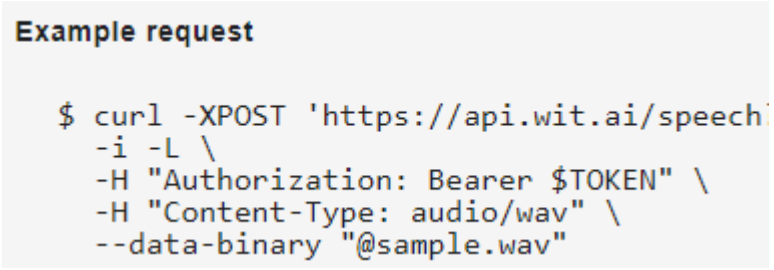
Fonte: Autoria própria (2022).

2.4.1.2 Wit AI

O *Wit AI* é uma ferramenta que utiliza o processamento de linguagem natural (PLN) para aplicações, capaz de transformar sentenças de áudio em dados estruturais. Possui uma interface simples e uma API de fácil aprendizado. É possível interagir com a ferramenta utilizando a voz ou linhas de textos e obter uma ação através dos comandos determinados a partir de predefinições programadas.

A aplicação da assistente de voz interage com a *Wit AI* através de comandos HTTP, como na figura 47 abaixo, sendo enviado o arquivo de áudio com o comando.

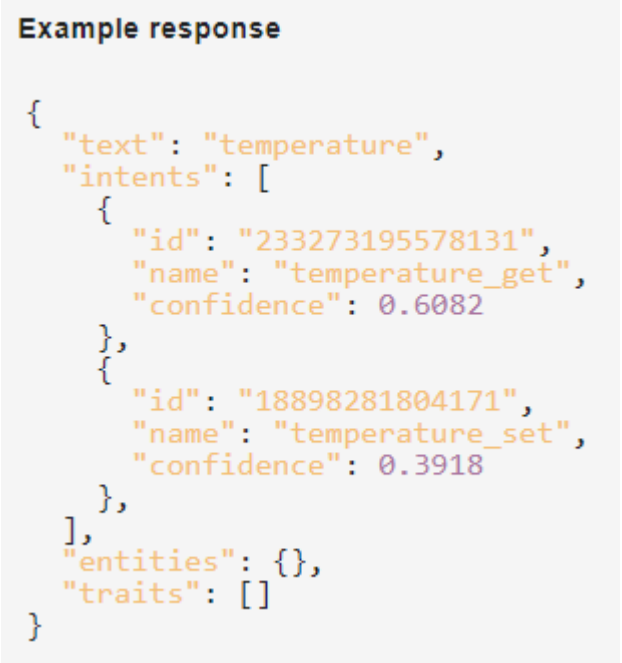
Figura 47 – Exemplo de requisição da Wit AI



Fonte: Autoria própria (2022).

O *payload* é recebido em formato JSON, como na figura 48 abaixo.

Figura 48 – Exemplo de resposta da Wit AI



Fonte: Autoria própria (2022).

Para a execução desses comandos, utilizou-se o componente *Axios* (cliente HTTP baseado em *Promises* para fazer requisições), como demonstra a figura 49 abaixo.

Figura 49 – Trecho de código da Assistente de Voz

```
public queryWitAi(name: string, room: string): Promise<any> {
    return new Promise((resolve, reject) => {
        let read = fs.readFileSync(RECORDINGS_PATH + name);
        axios.post(WIT_AI_URL, read, {
            headers: {
                'Authorization': 'Bearer ' + this.getNextToken(),
                'Content-Type': WIT_AI_CONTENT_TYPE
            }
        }).then((response: AxiosResponse) => {
```

Fonte: Autoria própria (2022).

A *Wit AI* tem uma interface de configuração muito prática, que permite configurar e obter informações identificadas nas gravações a partir dos conceitos abaixo.

2.4.1.2.1 Intents

Intents são as intenções do usuário — ou seja, o comando —, interpretadas pela inteligência artificial. Por exemplo: pesquisar uma música ou parar uma música. Estão representadas na figura 50 abaixo.

Figura 50 – Configuração na interface da Wit AI - Intents

Entity ↑↓	Roles	Intents
room_location	room_location	search_music, wit/stop_music
wit/search_query	music, search_query	search_music, wit/stop_music

Fonte: WIT AI (2022, p. 1)

2.4.1.2.2 Entities

Entities são as variáveis identificadas no comando. Para esta aplicação, existem duas *entities* que representam duas variáveis consideradas no comando. Estão representadas na figura 51 abaixo.

Figura 51 – Configuração na interface da Wit AI - Entities

Name ↑↓	Entities
search_music	wit/search_query:search_query, room_location:room_location, wit/search_query:music
wit/stop_music	room_location:room_location, wit/search_query:search_query

Fonte: WIT AI (2022, p. 1)

2.4.1.2.3 Utterances

Para o treinamento de comandos de voz, cada comando registrado gera uma *utterance*, que é o resultado do processamento de um áudio enviado para a *Wit AI*.

No treinamento, primeiro deve-se ouvir o áudio e atestar se o texto é válido. Exemplos estão representados nas figuras 52 e 53.

Figura 52 – Configuração na interface da Wit AI - Utterance exemplo 1

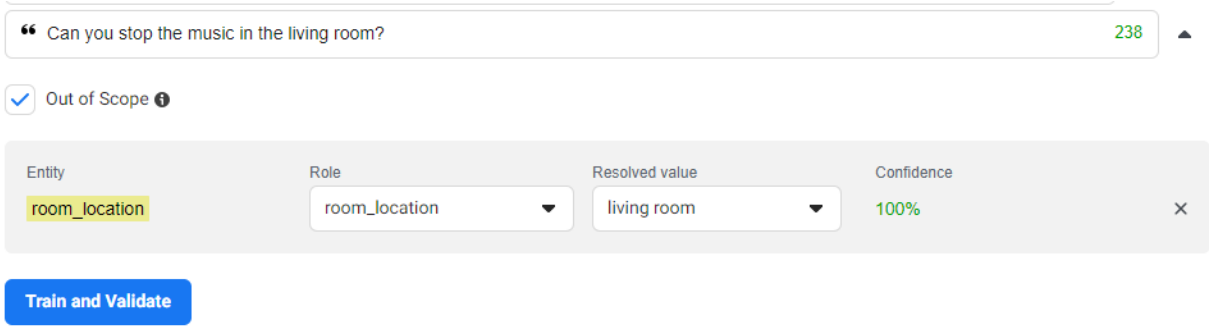
Fonte: WIT AI (2022, p. 1)

Figura 53 – Configuração na interface da Wit AI - Utterance exemplo 2

Fonte: WIT AI (2022, p. 1)

Feito isso, verifica-se a tomada de decisão quanto ao *intent* e à *entity*, como na figura 54.

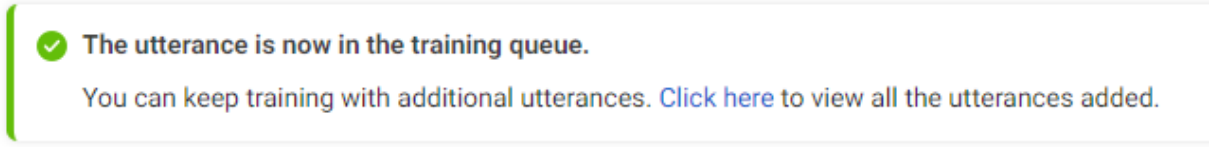
Figura 54 – Configuração na interface da Wit AI - Validar



Fonte: WIT AI (2022, p. 1)

Após a validação (e, se necessário, correção), a *utterance* é automaticamente adicionada à fila de treinamento, como na figura 55.

Figura 55 – Configuração na interface da Wit AI - Fila de treinamento



Fonte: WIT AI (2022, p. 1)

2.4.1.3 Integração e desenvolvimento

Após confirmar que a assistente de voz, que roda na Raspberry PI, funcionaria como esperado, desenvolveu-se a integração com o framework que roda na ESP32 e também outras implementações necessárias. Abaixo, a figura 56 mostra a assistente de voz e suas comunicações.

Figura 56 – Diagrama de funcionamento genérico da assistente de voz

Fonte: Autoria própria (2022).

A implementação da Assistente de Voz pode ser dividida em duas partes (entrada e saída de áudio) principais, descritas abaixo:

Saída de áudio:

Foi implementado, na assistente de voz, um serviço responsável por enviar, via REST (formato padronizado para modelos de requisições HTTP), ao *webhook* do servidor *Java*, um comando POST para iniciar ou parar a reprodução de um arquivo de áudio em um cômodo específico.

Esse serviço substituiu o serviço de áudio usado no primeiro projeto, relativo à prova de conceito, no que diz respeito à saída de áudio, representado na figura 57.

Figura 57 – Trecho de código da Assistente de Voz - playAudio

```
private playAudio(music: string, where: string) {
  this.server.playRequest(music, where)
  .then((response) => {
    // console.log('playRequest response: ', response);
  })
  .catch((error) => console.log('playAudio - playRequest error: ', error));
}
```

Fonte: Autoria própria (2022).

Entrada de áudio:

O áudio é obtido também através do *webhook* do *Stream Server*, no qual gravações são requisitadas periodicamente, em uma *thread* criada para essa função, executando o código abaixo, representado na figura 58.

Figura 58 – Trecho de código da Assistente de Voz - processVoiceCommand

```
private processVoiceCommand() {
    this.server.getRecording()
        .then((response: AxiosResponse) => {
            if(response && response.data && response.data !== '')
                this.recordingDoneSub.next(response.data);
        })
        .catch((error) => {
            console.error('Failed to retrieve recording from Stream Server. Connection lost.');
            this.recordingDoneSub.next([]);
        });
}
```

Fonte: Autoria própria (2022).

Essa requisição espera uma ou mais gravações, e cada gravação é representada por um objeto que contém o cômodo e nome do arquivo de gravação, como demonstrado abaixo, representado na figura 59.

Figura 59 – Trecho de código da Assistente de Voz - RecordingResponse

```
export interface RecordingResponse {
    room: string,
    file: string
}
```

Fonte: Autoria própria (2022).

Existe um algoritmo implementado, no *Stream Server*, garantindo a sincronia entre o que está sendo gravado pelo ESP32 e o que está sendo enviado pela IA, para evitar *gaps* (consequentemente, *delays* na interpretação/execução do comando) por reconexão ou latência.

Diferente da prova de conceito, esta implementação passa a ser responsável pela gravação, sendo removido o acesso direto da IA aos dispositivos de entradas de áudio.

2.4.1.4 Suporte para múltiplos clientes

Localização:

A definição do local para onde o comando está sendo direcionado é feita através da implementação da *entity: room_location*.

Essa variável textual é usada para buscar o *socket* correspondente à conexão daquele cômodo. A busca é feita no objeto de configuração, criado na inicialização com base nas entradas fornecidas no momento de conexão (como consta no tópico 2.3.3 Suporte para Múltiplos Clientes).

Com o intuito de identificar se o cômodo pesquisado através do comando está conectado, ao receber o valor da entidade *room_location* como resposta do servidor da *Wit AI*, é utilizado um filtro de comparação.

Na implementação do filtro, o valor de *data.location* corresponde à entidade e o valor de *conn.room* corresponde ao cliente conectado. Essas variáveis são comparadas e devem ser iguais.

A parte do código que contempla essa verificação está na figura 60 abaixo.

Figura 60 – Trecho de código da Assistente de Voz - mutex

```
while(this.mutex) {};
this.mutex = true;
  let match: AudioInputConnection[] = this.connections.filter(conn => {
    return conn.room === data.location;
  });
```

Fonte: Autoria própria (2022).

Controle:

A maior parte do controle da localização é realizado pelo *Stream Server*. O que diz respeito à assistente de voz é o consumo de recursos do servidor *Wit AI* (*Facebook*).

O uso gratuito da *Wit AI* é limitado a 60 *requests* por minuto. Como, para cada ponto, são enviados *requests* a cada 1,1 segundo (totalizando 54 *requests*/minuto), não seria possível a execução com mais de um ponto.

Para contornar esse problema, criou-se um clone da aplicação da *Wit AI* para cada ponto. Para a apresentação, na qual três pontos seriam utilizados, foram feitos dois clones (além do original).

Assim, executa-se um *round-robin* (2020) antes do envio de cada *request* aos servidores do *Facebook*, alternando entre os três clones da aplicação, como podemos ver nas figuras 61 e 62.

Figura 61 – Trecho de código da Assistente de Voz - queryWitAi

```
public queryWitAi(name: string): Promise<any> {
    return new Promise((resolve, reject) => {
        let read = fs.readFileSync(RECORDINGS_PATH + name);
        axios.post(WIT_AI_URL, read, {
            headers: {
                'Authorization': 'Bearer ' + this.getNextToken(),
                'Content-Type': WIT_AI_CONTENT_TYPE
            }
```

Fonte: Autoria própria (2022).

Figura 62 – Trecho de código da Assistente de Voz - getNextToken

```
private getNextToken(): string {
    if(this.count > 2)
        this.count = 0;
    switch (this.count++) {
        case 0:
            return WIT_AI_TOKEN1;
        case 1:
            return WIT_AI_TOKEN2;
        case 2:
            return WIT_AI_TOKEN3;
        default:
            this.count = 0;
            return WIT_AI_TOKEN1;
    }
}
```

Fonte: Autoria própria (2022).

2.4.1.5 YouTube data API

Para obter a URL do vídeo escolhido, a aplicação se comunica com a API de dados do YouTube através de um componente *NPM* (2020) chamado *youtube-search* (NPM, 2021). Exemplo na figura 63.

Figura 63 – Exemplo de snippet de uso - youtubeSearch

```
import * as youtubeSearch from "youtube-search";

var opts: youtubeSearch.YouTubeSearchOptions = {
  maxResults: 10,
  key: "yourkey"
};

youtubeSearch("jsconf", opts, (err, results) => {
  if(err) return console.log(err);

  console.dir(results);
});
```

Fonte: Autoria própria (2022).

O atributo *yourkey* é a chave de acesso garantida à conta criada em *Google Identity Platform* (GOOGLE, 2021) utilizada para esse sistema, o *youtubeSearch* é o método utilizado para obter a URL e *jsconf* é a *string* substituída pelo nome da música buscada.

Os resultados obtidos são serializados neste formato da figura 64.

Figura 64 – Interface de resultado da pesquisa no Youtube

```
export interface YouTubeSearchResults {
  id: string;
  link: string;
  kind: string;
  publishedAt: string;
  channelTitle: string;
  channelId: string;
  title: string;
  description: string;
  thumbnails: YouTubeSearchResultThumbnails;
}
```

Fonte: Autoria própria (2022).

Cada atributo dessa resposta está documentado na referência da API. Apenas duas informações são utilizadas.

1. *link*: representa a URL do vídeo.
2. *kind*: especifica o tipo do resultado, utilizado no filtro demonstrado na figura 65 abaixo, excluindo os resultados do tipo canal e *playlist*, pois correspondem às entradas que não contêm áudio.

Figura 65 – Trecho de código da Assistente de Voz - análise de pesquisa

```
if(results) {
  // console.log('search on youtube done. analyzing results ...');
  for (let i = 0; i < results.length; i++) {
    // if it is video (not a channel)
    if(results[i].kind === 'youtube#video') {
      this.download(results[i].link, cmd);
      return;
    }
  }
}
```

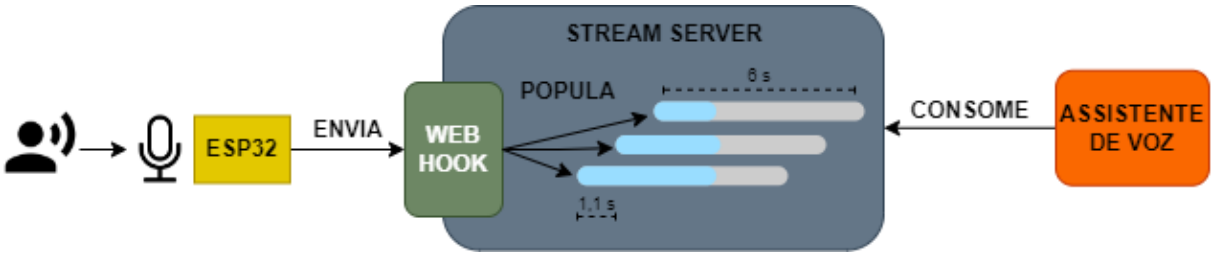
Fonte: Autoria própria (2022).

2.4.2 Stream Server

Para o desenvolvimento do servidor do sistema, é utilizado o projeto do *Stream Server*. Essa aplicação se comunica com cada cliente e com a assistente de voz. As duas principais funções deste servidor são as de entrada e saída de áudio.

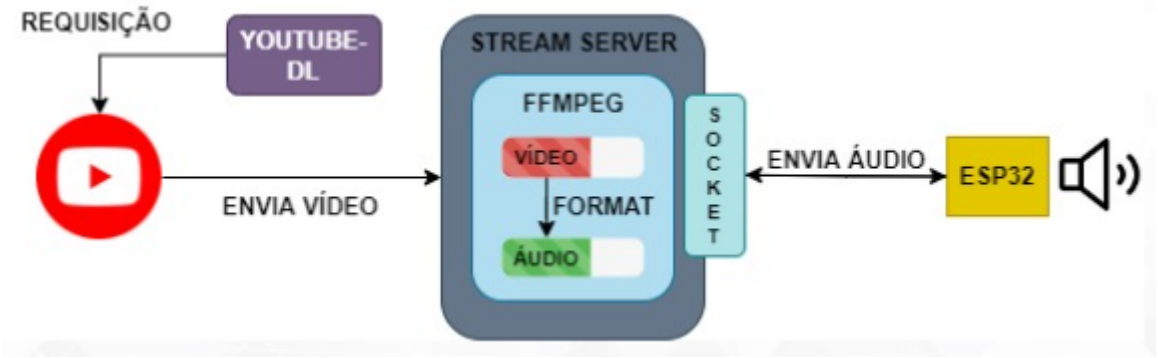
O procedimento básico dessas duas funções está ilustrado nas figuras 66 e 67 abaixo, respectivamente.

Figura 66 – Diagrama de funcionamento da entrada de áudio no SS



Fonte: Autoria própria (2022).

Figura 67 – Diagrama de funcionamento da saída de áudio no SS



Fonte: Autoria própria (2022).

2.4.2.1 Rest API

O *Stream Server* tem uma API para receber comandos da assistente de voz e clientes ESP32.

A classe *RestServiceAPI* dentro do *Stream Server* contém as requisições utilizadas na comunicação entre o servidor e os clientes.

O *endpoint POST /play* tem como função iniciar a reprodução da música no cliente, informando o cômodo e o arquivo de áudio, representado na figura 68.

Figura 68 – Endpoint */play* no *Stream Server*

```
@PostMapping("/play")
public String play(@RequestBody PlayRequestDTO playRequestDTO) {
    new Thread(() -> {
        conHandler.decode(playRequestDTO.getFile(), playRequestDTO.getWhere());
        conHandler.play(playRequestDTO.getFile(), playRequestDTO.getWhere());
    }).start();
    return "playing music file: " + playRequestDTO.toString();
}
```

Fonte: Autoria própria (2022).

O objeto serializado está na figura 69 abaixo.

Figura 69 – DTO da requisição de *play*

```
public class PlayRequestDTO

    String file;
    String where;
```

Fonte: Autoria própria (2022).

O *endpoint POST /stop* tem como função parar a reprodução da música no cliente com a configuração do cômodo informado, representado na figura 70.

Figura 70 – Endpoint *stop* no *Stream Server*

```
@PostMapping("/stop")
public String stop(@RequestBody PlayRequestDTO playRequestDTO) {
    new Thread(() -> {
        conHandler.stop(playRequestDTO.getWhere());
    }).start();
    return "stopping music on: " + playRequestDTO.getWhere();
}
```

Fonte: Autoria própria (2022).

O *endpoint POST /adc_samples* tem como função registrar um pacote de áudio gravado pelo microfone no cliente com a configuração do cômodo informado.

Figura 71 – Endpoint */adc_samples* no *Stream Server*

```
@PostMapping("/adc_samples")
public String record(@RequestBody byte [] bytes,
        @RequestHeader("room-name") String room) {
    new Thread(() -> {
        this.recHandler.processInputStream(bytes, room);
    }).start();
    return "OK";
}
```

Fonte: Autoria própria (2022).

O *endpoint GET /getrecording* tem como função retornar o endereço de uma gravação de áudio com 6,6 segundos. As gravações são criadas através do processamento dos pacotes recebidos no *endpoint POST /adc_samples* (citado acima na figura 71). O endpoint está na figura 72.

Figura 72 – Endpoint */getrecording* no Stream Server

```
@GetMapping("/getrecording")
public synchronized String getRecording() { return this.recHandler.getRecording(); }
```

Fonte: Autoria própria (2022).

Além desses *endpoints*, outros como *POST /connection* e *GET /keepalive* foram criados e estão explicados na seção de suporte para múltiplos clientes e *watchdog*. Todos os métodos desenvolvidos para as requisições em *RestServiceAPI* estão na classe *ConnectionHandler* e *RecordingHandler*.

Na classe *ConnectionHandler*, o método *start* inicia o *socket* do servidor na porta 4444 e espera a conexão com o cliente. O método está na figura 73.

Figura 73 – Trecho de código do *Stream Server* - start

```
@PostConstruct
private void start() {
    try {
        this.listener = new ServerSocket( port: 4444);
        this.running = true;
        new Thread(new ListenThread()).start();
    } catch (Exception e) {
        System.out.println("setup() failed: ");
        e.printStackTrace();
    }
}
```

Fonte: Autoria própria (2022).

O método *listen* aguarda conexões no *socket* que foi iniciado através do método *start*. Está representado na figura 74.

Figura 74 – Trecho de código do *Stream Server* - listen

```
private void listen() {
    while (this.running) {
        try {
            Socket socket = listener.accept();
            System.out.println("New client connected: " + socket.getInetAddress());
            SocketHandler sh = new SocketHandler(socket, config.getMediaPath());
            this.sockets.put(count++, sh);
            new Thread(sh).start();
        } catch (Exception e) {
            System.out.println("listen() failed: ");
            e.printStackTrace();
        }
    }
    System.out.println("server shutdown");
}
```

Fonte: Autoria própria (2022).

O método *play* inicia a reprodução do arquivo de música identificado pelo parâmetro *file* no cômodo especificado pelo parâmetro *room,* representado na figura 75.

Figura 75 – Trecho de código do *Stream Server* - play room

```
public void play(String file, String room) {
    Integer where = config.getId(room);
    System.out.println("Play command for file [" + file + "] @ [" + room + "].");
    if (this.sockets.containsKey(where))
        this.sockets.get(where).play(file);
    else
        System.out.println("There's no active connection on socket: " + where);
}
```

Fonte: Autoria própria (2022).

O método *play* (utilizado acima pelo objeto do vetor *sockets*) está descrito abaixo e tem como função fazer a conversão do formato do arquivo de *.opus* para *.wav*. O arquivo *.opus* é um arquivo de áudio digital codificado com o *Opus* (2020) (*codec* de compressão de áudio aberto), representado na figura 76.

Figura 76 – Trecho de código do *Stream Server* - play

```
public void play(String fileName) {
    try {
        String name = this.renameOpus2Wav(fileName);
        System.out.println("Playing " + name);
        fis = new FileInputStream(Util.checkFile( filename: this.mediaPath + name));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Fonte: Autoria própria (2022).

O método *decode* faz a decodificação do arquivo de música recebido pelo *FFmpeg*. Esse método inicia uma *thread* que formata o *stream* de áudio recebido do *YouTube* para o formato *.wav*, representado na figura 77.

Figura 77 – Trecho de código do *Stream Server* - decode room

```
public void decode(String file, String room) {
    Integer where = config.getId(room);
    System.out.println("Decode command for file [" + file + "] @ [" + room + "].");
    if (this.sockets.containsKey(where))
        mh.decode(file);
    else
        System.out.println("There's no active connection on socket: " + where);
}
```

Fonte: Autoria própria (2022).

O método *stop* para a música que está sendo reproduzida no cômodo especificado, representado na figura 78.

Figura 78 – Trecho de código do *Stream Server* - stop

```
public void stop(String room) {
    System.out.println("Stop command @ [" + room + "].");
    Integer where = config.getId(room);
    if (this.sockets.containsKey(where))
        this.sockets.get(where).stop();
    else
        System.out.println("There's no active connection on socket: " + where);
}
```

Fonte: Autoria própria (2022).

O método *setConnection* registra a identificação do cômodo ao endereço IP do cliente conectado via *socket*, representado na figura 79.

Figura 79 – Trecho de código do *Stream Server* - setConnection

```
public void setConnection(String room, String ip) {
    for(Entry<Integer, SocketHandler> et : sockets.entrySet())
        if(et.getValue().getIp().replaceFirst( regex: "/", replacement: "").equals(ip))
            config.addConfiguration(room, et.getKey());
}
```

Fonte: Autoria própria (2022).

2.4.2.2 Conexão

A conexão entre o servidor e o cliente é implementada através de *sockets*. O servidor utiliza os métodos para inicializar o *socket* na porta determinada e, assim, cada cliente consegue solicitar a conexão.

Os métodos como *start*, *listen* e *setConnection*, já mencionados, estabelecem essas conexões.

2.4.2.3 Watchdog

Como mencionado na seção Watchdog do firmware (2.3.4), criou-se um *endpoint*, do tipo GET, com uma resposta simples. O conteúdo da resposta não interessa, apenas o *status code*. O *endpoint* foi implementado como na figura 80.

Figura 80 – Endpoint */keepalive* no *Stream Server*

```
@GetMapping("/keepalive")
public String keepAlive() { return "Ok"; }
```

Fonte: Autoria própria (2022).

2.4.2.4 Tratamento de dados

Foi implementado, no *Stream Server*, um algoritmo de conversão de arquivos de áudio, pois o formato do dado recebido do *YouTube* não é o ideal, porque na implementação atual não pode ser convertido diretamente na ESP32.

O problema aqui é que essa conversão precisa ser feita em tempo real, pois não é aceitável esperar o download do arquivo completo para iniciar a reprodução.

Toda vez que um pedido de música é recebido, o servidor, primeiramente, inicia uma *thread* de decodificação. Logo após, sem aguardar o término, inicia a *thread* de reprodução. Pode-se observar, na figura 81 abaixo, o algoritmo de decodificação implementado, onde inicialmente trata-se o nome do arquivo (alteração da extensão) e depois, periodicamente, executa-se a conversão dos dados para o formato desejado.

Figura 81 – Trecho de código do *Stream Server* - decode

```
private void decode(String filename) throws Exception {
    String inputWithPart = mediaPath.concat(filename).concat(partExt)
            .replaceAll("\n", "").replaceAll("\r", "");
    String inputComplete = mediaPath.concat(filename)
            .replaceAll("\n", "").replaceAll("\r", "");
    String output = mediaPath.concat(filename).replaceAll(inputExt, outputExt)
            .replaceAll("\n", "").replaceAll("\r", "");

    File f = Util.checkFile(inputWithPart);
    long lastSize = 0;
    long len = 0;
    int rc = 0;
    int timeoutInSeconds = 10;
    long timeout = System.currentTimeMillis() + timeoutInSeconds*1000;
    while(this.ffmpeg(inputWithPart, output) != 0) {
        if(System.currentTimeMillis() >= timeout)
            throw new Exception("FFMPEG failed to start in " + timeoutInSeconds + " seconds!");
    }
    do {
        rc = this.ffmpeg(inputWithPart, output);
        if (rc < 0)
            throw new Exception("FFMPEG error during decoding!");
        Util.delay(3000);
        len = f.length();
        if (rc == 0 && len == lastSize) {
            System.out.println(
                    "The download has finished. Maybe it failed to rename due to 'file already in use' error.");
            return;
        }
        lastSize = len;
    } while (rc != 1);

    if (this.ffmpeg(inputComplete, output) > 0)
        throw new Exception("FFMPEG error on last decoding!");
}
```

Fonte: Autoria própria (2022).

Figura 82 – Trecho de código do *Stream Server* - ffmpeg

```
private int ffmpeg(String input, String output) {
    try {
        Process p = new ProcessBuilder(ffmpegPath, "-i", input, "-ar", sampleRate, "-y", output).start();
        return p.waitFor();
    } catch (IOException | InterruptedException e) {
        e.printStackTrace();
    }
    return -1;
}
```

Fonte: Autoria própria (2022).

A conversão é feita pela ferramenta *FFmpeg*, executada via *ProcessBuilder* (ORACLE, 2021), representado na figura 82.

Na parte do servidor, a classe *ConfigurationManager* contém as configurações para o arquivo *ffmpeg.exe*, a taxa de amostragem do áudio e as extensões dos arquivos envolvidos. As configurações estão na figura 83.

Figura 83 – Configuração de constantes no *Stream Server*

```
private Map<String, Integer> config = new HashMap<String, Integer>();
private final String ffmpegPath = "C:\\Users\\Filipe Itonaga\\TCC\\tcc\\voice-assistant\\backend\\tools\\ffmpeg\\bin\\ffmpeg.exe";
private final String mediaPath = "C:\\Users\\Filipe Itonaga\\TCC\\tcc\\voice-assistant\\backend\\";
private final String sampleRate = "16000";
private final String inputExt = ".webm";
private final String outputExt = ".wav";
private final String partExt = ".part";
```

Fonte: Autoria própria (2022).

2.4.2.5 Suporte para múltiplos clientes

No *Stream Server*, para controle dos *sockets*, o *endpoint POST /connection* é utilizado. O cômodo e IP recebidos são associados, como nas figuras 84 e 85 abaixo.

Figura 84 – Endpoint */connection* no *Stream Server*

```
@PostMapping("/connection")
public String connect(@RequestBody ConnectRequestDTO body) {
    new Thread(() -> {
        String room = body.getRoom();
        String ip = body.getIp();
        System.out.println("Setting [" + room + "] for [" + ip + "].");
        this.conHandler.setConnection(room, ip);
    }).start();
    return "Ok";
}
```

Fonte: Autoria própria (2022).

Figura 85 – DTO da requisição de conexão no *Stream Server*

```
public class ConnectRequestDTO {

    private String room;
    private String ip;
```

Fonte: Autoria própria (2022).

Dessa forma, consegue-se detectar reconexões (caso o cliente seja reconectado) e desalocar recursos alocados na sessão anterior.

Ainda, essa configuração feita na conexão é utilizada para verificar se existe de fato um cliente conectado no cômodo passado pelo comando.

2.4.3 Limpando arquivos antigos

Durante o desenvolvimento do projeto, percebeu-se a necessidade de limpar o espaço utilizado, ou seja, deletar antigos arquivos de mídia, como gravações e músicas. As gravações são excluídas periodicamente, enquanto as músicas são deletadas na inicialização e no comando de parar: *stop()*.

Na figura 86, é possível ver o trecho específico à deleção das gravações, com um agendamento de 10 segundos e apagando gravações mais antigas que 20 segundos.

Figura 86 – Trecho de código do *Stream Server* - cleanOldRecordings

```
@Scheduled(fixedDelay = 10*1000) // 10 seconds
public void cleanOldRecordings() {
    Long now = new Date().getTime();
    File recordingFolder = new File(config.getRecordingsPath());
    if(!recordingFolder.isDirectory())
        return;
    for(File file : recordingFolder.listFiles()) {
        if(now - this.extractRecordingDate(file.getName()) > 20000) {
            file.delete();
        }
    }
}
```

Fonte: Autoria própria (2022).

Para as músicas baixadas, existem dois momentos de deleção, figuras 87 e 88.

1) quando a aplicação é inicializada; e 2) quando acontece um comando *stop()*:

1)

Figura 87 – Trecho de código do *Stream Server* - cleanFilesOnStartup

```
@EventListener(ApplicationReadyEvent.class)
public void cleanFilesOnStartup() {
    logger.info("Cleaning files on startup");
    File mediaFolder = new File (config.getMediaPath());
    if (!mediaFolder.isDirectory())
        return;
    for (File file1 : mediaFolder.listFiles()) {
        // System.out.println("cleanOldRecordings() - Checking recording date: " + file.getName());
        if (file1.getName().contains(".wav") || file1.getName().contains(".webm")) {
            logger.info("deleteMedia() - Deleting Song: " + file1.getName());
            file1.delete();
        }
    }
}
```

Fonte: Autoria própria (2022).

2)

Figura 88 – Trecho de código do *Stream Server* - deleteMedia

```
private void deleteMedia() {
    String nameCurrentSongClean = nameCurrentSong.replace( target: ".wav",  replacement: "");
    File mediaFolder = new File(mediaPath);
    if(!mediaFolder.isDirectory())
        return;
    for(File file1 : mediaFolder.listFiles()) {
        if(file1.getName().contains(nameCurrentSongClean)) {
            logger.info("deleteMedia() - Deleting Song: " + file1.getName());
            file1.delete();
        }
    }
}
```

Fonte: Autoria própria (2022).

Durante a deleção, foram encontrados alguns problemas em tempo de execução que, apesar de não serem mudanças essenciais para o funcionamento, atrapalhavam *logs* e a visualização:

1) A *thread* no servidor que faz o *decode* do arquivo de áudio para ser enviado ao ESP32 retorna uma exceção por não encontrar o arquivo de música. Está na figura 89.

Solução: salvar a *thread* em um mapa com o cômodo como chave e encerrar essa *thread* no *stop()* executado naquele cômodo.

Figura 89 – Trecho de código do *Stream Server* - remover thread

```
if (this.sockets.containsKey(where)) {
    if (mh.decodeThreadMap.containsKey(room)) {
        if (mh.decodeThreadMap.get(room).isAlive())
            mh.decodeThreadMap.get(room).stop();
        mh.decodeThreadMap.remove(room);
    }
    this.sockets.get(where).stop();
}
```

Fonte: Autoria própria (2022).

2) A *thread* na IA que baixa a música em execução retorna uma exceção, caso o download ainda esteja em execução. Está na figura 90.

Solução: salvar o PID do processo do *YouTube DL* em um mapa, seguindo o mesmo conceito da solução acima.

Figura 90 – Trecho de código da Assistente de Voz

```
public stopDownload(room: string) {
  let pid = this.roomPidMap.get(room);
  if(pid && pid !== -1) {
    process.kill(pid, 'SIGINT');
    this.roomPidMap.set(room, -1);
  }
}
```

Fonte: Autoria própria (2022).

2.4.3.1 Problema de performance encontrado

Quando executados testes com três clientes conectados, notou-se que um chiado alto começa a surgir com o passar do tempo.

Após várias tentativas de corrigir ou reduzir o ruído, percebeu-se que o algoritmo de transferência e reprodução dos dados de áudio deveria ser trocado por um *buffer* circular mais eficiente.

Foi decidido que, para este projeto acadêmico, isso não seria feito.

2.5 Ferramentas de desenvolvimento de projetos

Para o desenvolvimento e gerenciamento do projeto, foram utilizadas algumas ferramentas de uso livre (não comercial). Será explicado abaixo as duas mais importantes.

Para a organização das tarefas e acompanhamento do progresso, utilizou-se a *Kanban Board* da ferramenta *ClickUp* (2021).

Para o compartilhamento e versionamento de código, utilizou-se o *GitHub* (2021).

Para o compartilhamento e armazenamento de documentos e arquivos, utilizou-se o *Google Drive*.

Para o desenvolvimento do software em *Java*, foram utilizadas as IDEs *Eclipse* (2020) e *IntelliJ* (JETBRAINS, 2021) (dependendo da preferência do aluno).

Para o desenvolvimento do software em *Typescript*, utilizou-se o *Visual Studio Code* (MICROSOFT, 2021).

Para o desenvolvimento do *firmware* em C, utilizou-se o *Arduino Studio* (2020).

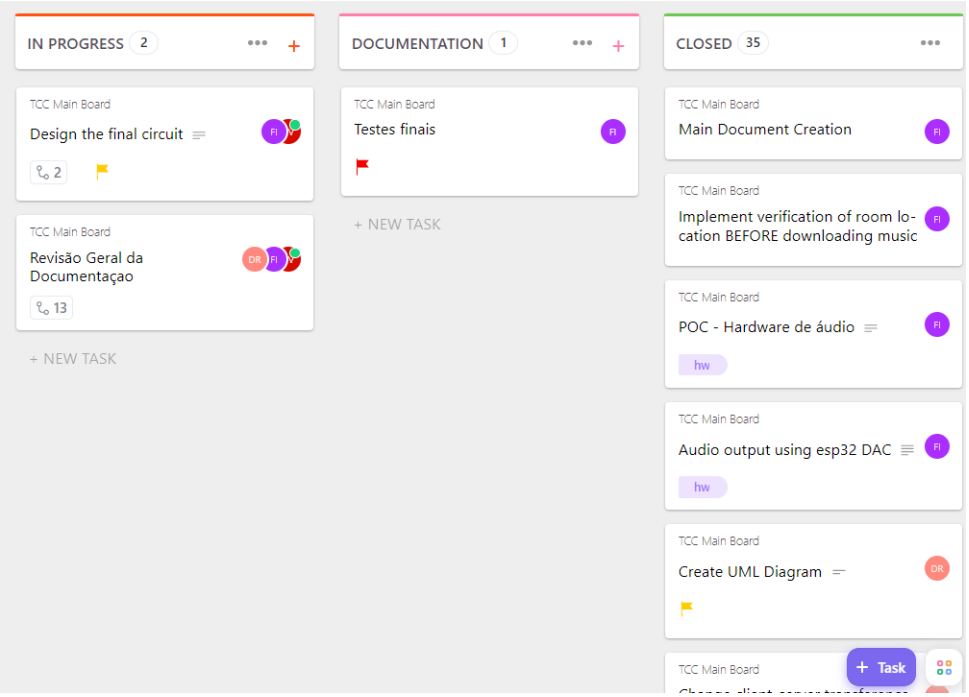
Para a comunicação entre os alunos, foram utilizados o *Discord* (2020) e o *WhatsApp* (2020).

Para a comunicação com o professor orientador, utilizou-se o *Google Meets* (GOOGLE, 2021).

2.5.1 Clickup

A aplicação foi usada para que a equipe tivesse maior controle sobre as atividades correntes, planejadas e já resolvidas, além de poder adicionar novas demandas conforme a necessidade no decorrer do projeto. Essa ferramenta permitiu acompanhar o progresso do projeto, individualmente e como um todo, além de adicionar uma forma de fácil visualização ao professor orientador. Representado na figura 91.

Figura 91 – Resumo do quadro do *Clickup*



Fonte: Click Up (2021, p. 1)

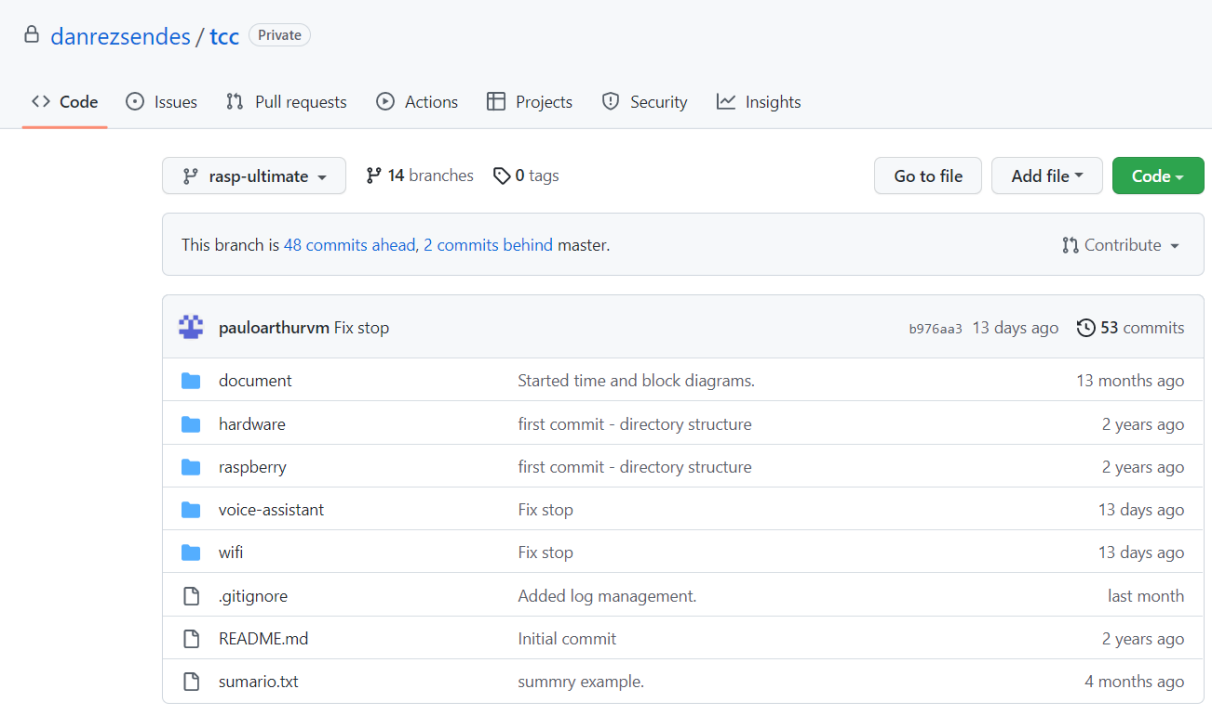
Em geral, os passos para cada tarefa seguiam o mesmo fluxo:

- Criar a *task* e deixá-la aberta (*Open*). Indica o que deve ser feito.
- Selecionar para começar as mais importantes (*Pending*). Indica as próximas tarefas.
- Selecionar algumas mais demoradas e não essenciais para melhorias (*Improvement*). Essas eram consumidas quando havia sobra de tempo.
- Iniciar uma tarefa e movê-la para *In Progress*. Indica o que está sendo trabalhado.
- Ao finalizar, mover para *In Review*. Indica revisão (por outro aluno) pendente.
- Ao revisar, mover para *Completed***.** Indica o fim da tarefa em si.
- Mover para *Documentation*. Indica documentação pendente por parte do desenvolvedor.
- Mover para *In Review*. Indica revisão da documentação pendente por parte de outro aluno.
- Quando tudo está implementado, documentado e revisado, move-se a tarefa para *Closed*. Indica finalização integral da tarefa.

2.5.2 Github

Para versionamento de código, foi utilizado o *Git* (2020) e, junto dele, o *Github*, como plataforma de repositório. Durante o projeto, foram criadas *branchs* de acordo com a necessidade das tarefas, para compartilhamento de código sem "contaminação" do *branch* principal. Representado na figura 92.

Figura 92 – Repositório do projeto final no *Github*



Fonte: SOUZA, D (2020, p. 1)

3 RESULTADOS

Testes foram feitos com três clientes conectados simultaneamente no mesmo servidor, conectados via Wi-Fi em uma internet de residência de 300 MB de velocidade. Quando medida (com medidor de velocidade online) no servidor, a velocidade de download e upload eram de 41 Mbps e 45 Mbps, respectivamente.

Para simulação de caso real, cada cliente foi posicionado em um cômodo da residência, devidamente identificado. Por exemplo: *bedroom*, *kitchen* e *living room*.

O treinamento foi executado algumas vezes durante um período de quatro a seis semanas.

Em resumo, os dados de teste são:

- Número de clientes: 3;
- Cômodos utilizados: *bedroom*, *kitchen* e *living room;*
- Período de treinamento da *Wit AI*: quatro a seis semanas (como explicado na seção 2.4.1.2.3 *Utterances*);
- Velocidade da internet: 41 MBps para download e 45 MBps para upload.

Os resultados (em função do tempo) encontrados, medidos em segundos, com cinco amostras cada, estão na tabela 1.

Tabela 1 – Resultados em função do tempo

Tempo de resposta do(a)	**T1 (s)**	**T2 (s)**	**T3 (s)**	**T4 (s)**	**T5 (s)**	**Média**
Interpretação do comando de *start*	8.0	4.2	7.0	2.3	4.4	5.2
Execução do comando de *start*	10.0	7.3	6.0	7.1	5.9	7.3
Interpretação e execução do comando de *stop*	3.0	4.0	3.0	3.9	3.6	3.5
Interpretação do comando de *start* cruzado	6.7	4.3	6.0	6.1	6.2	5.9
Execução do comando de *start* cruzado	7.5	7.0	7.2	7.0	6.7	7.1
Interpretação e execução do comando de *stop* cruzado	7.5	4.8	5.8	8.0	4.0	6.0

Tempo de resposta do(a)	T1 (s)	T2 (s)	T3 (s)	T4 (s)	T5 (s)	Média
Interpretação do comando de troca de música	6.4	6.1	5.7	5.0	3.2	5.3
Execução do comando de troca de música	9.4	7.4	6.5	8.3	6.9	7.7
Startup do *Stream Server*	10.3	10.3	10.3	10.3	10.3	10.3
Startup do assistente de voz	25.9	25.9	25.9	25.9	25.9	25.9
Conexão de um cliente no servidor	17.7	13.2	2.0	12.6	4.0	9.9

Fonte: Tabela do autor

Os resultados (em função do número de tentativas) encontrados, com cinco amostras cada, estão na tabela 2.

Tabela 2 – Resultados em função do número de tentativas

Número de tentativas do	T1	T2	T3	T4	T5	Média
Comando *start*	2	1	1	1	1	1.2
Comando *stop*	1	1	2	3	3	2
Comando *start* cruzado	1	2	1	1	2	1.4
Comando *stop* cruzado	5	2	3	6	1	3.4
Comando de troca de música	1	1	2	2	5	2.2

Fonte: Tabela do autor

Os quadrados vermelhos significam que uma ou mais entidades estavam erradas (em relação ao que foi falado). Os verdes significam que ambas as entidades estavam certas.

Por exemplo, alguns casos de erro:

1. Pediu-se “Play Tim Maia in the kitchen”.
   Interpretou-se “Play John Mayer in the kitchen”.
2. Pediu-se “Play Charlie Brown Jr in the bedroom”

Interpretou-se “Play James Brown in the bathroom”

3. Pediu-se “Play Michael Jackson in the bedroom”

   Interpretou-se “Play Michael Jackson in the bathroom”

Em relação ao consumo de recursos de máquina e rede, os resultados podem ser vistos abaixo, em dois tipos de cenários de uso:

Cenário controlado:

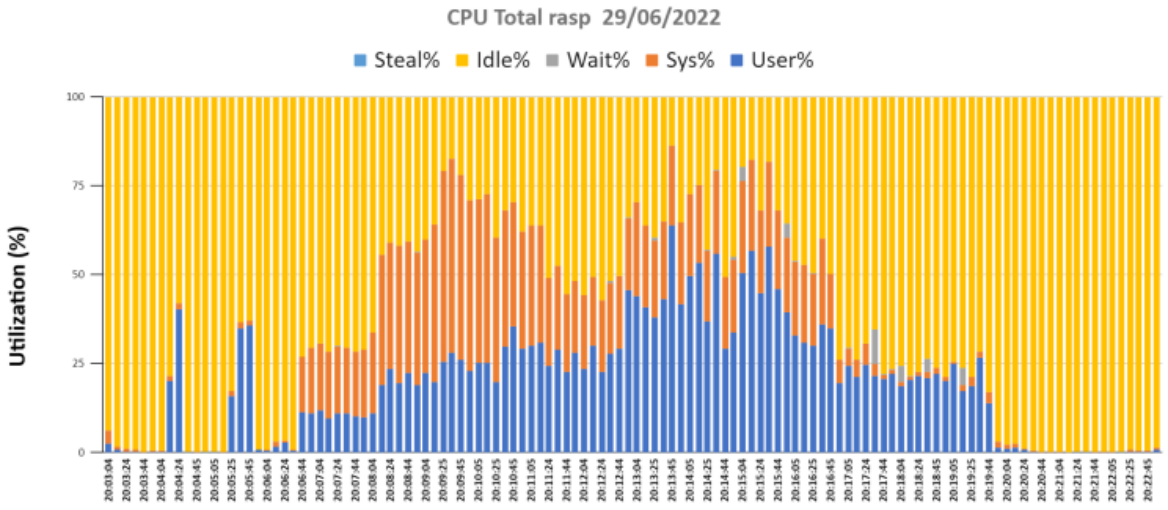
Neste cenário, as aplicações *Java* e *Node* foram iniciadas, respectivamente, às 20:04 e 20:05.

Os clientes foram conectados nos minutos 20:06, 20:08 e 20:09.

Os *streams* se iniciaram às 20:10, 20:12 e 20:15. Nesse último momento, ocorreu uma possível queda de rede e os clientes foram desconectados. A desconexão dos clientes foi às 20:15 e as aplicações foram finalizadas às 20:19.

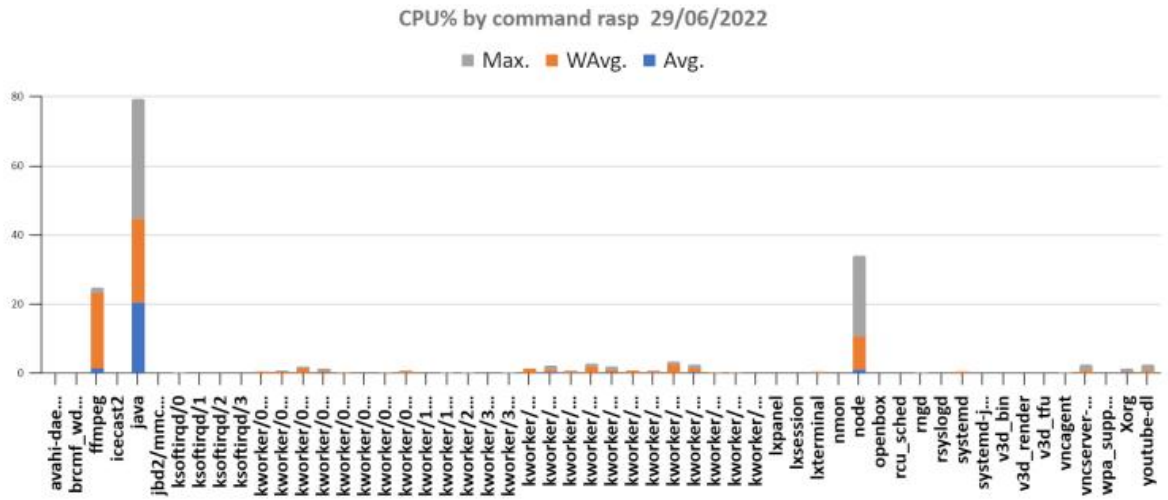
Consumo de CPU:

Figura 93 – Consumo de CPU x tempo na *Raspberry Pi*



Fonte: nmon analyser

Figura 94 – Consumo de CPU x comando na *Raspberry Pi*



Fonte: nmon analyser

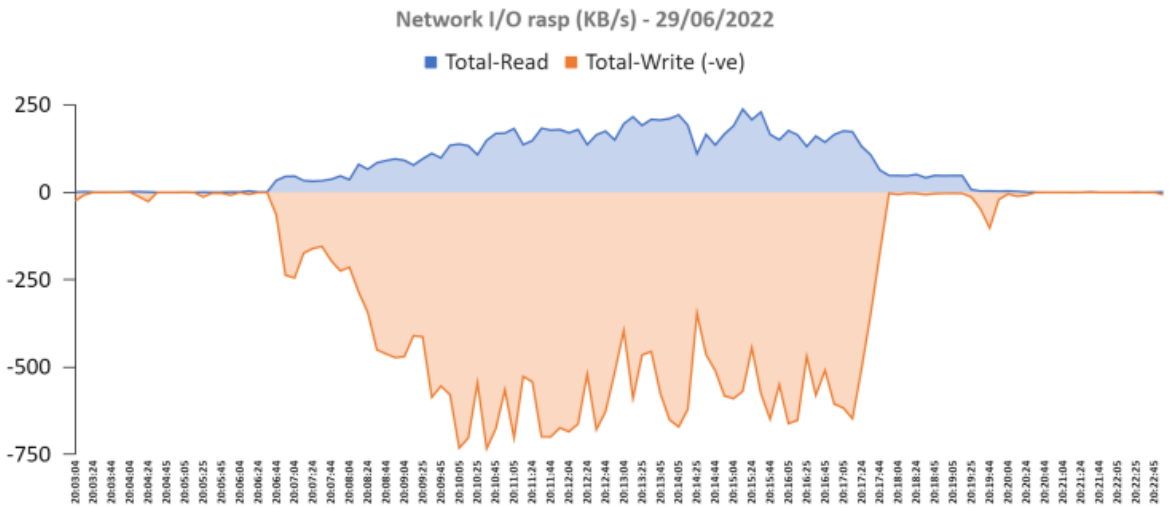
No momento de inicialização das aplicações, nota-se dois picos de consumo de processamento (aproximadamente 40% de uso da CPU).

Nos momentos de conexão dos clientes, nota-se uma subida na reta, indicando um consumo de CPU (aproximadamente 30% de uso da CPU por conexão).

Durante o uso, nota-se aumentos do consumo durante a reprodução. Ao desconectar os clientes, percebe-se uma grande redução no consumo da CPU. O consumo tende a zero após o encerramento das aplicações. Estão representados nas figuras 93 e 94 acima.

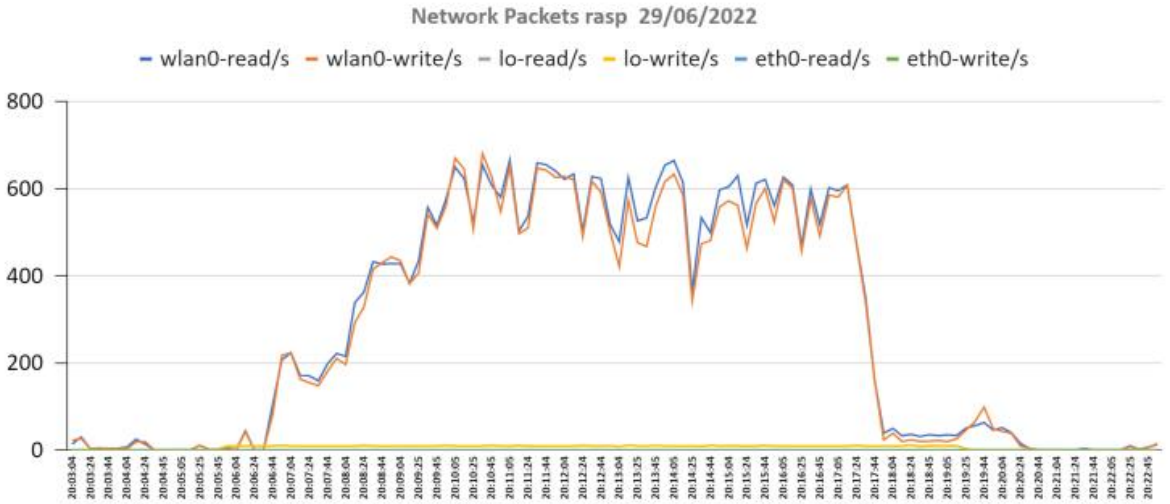
Consumo de rede:

Figura 95 – Consumo de Rede x tempo na *Raspberry Pi*



Fonte: nmon analyser

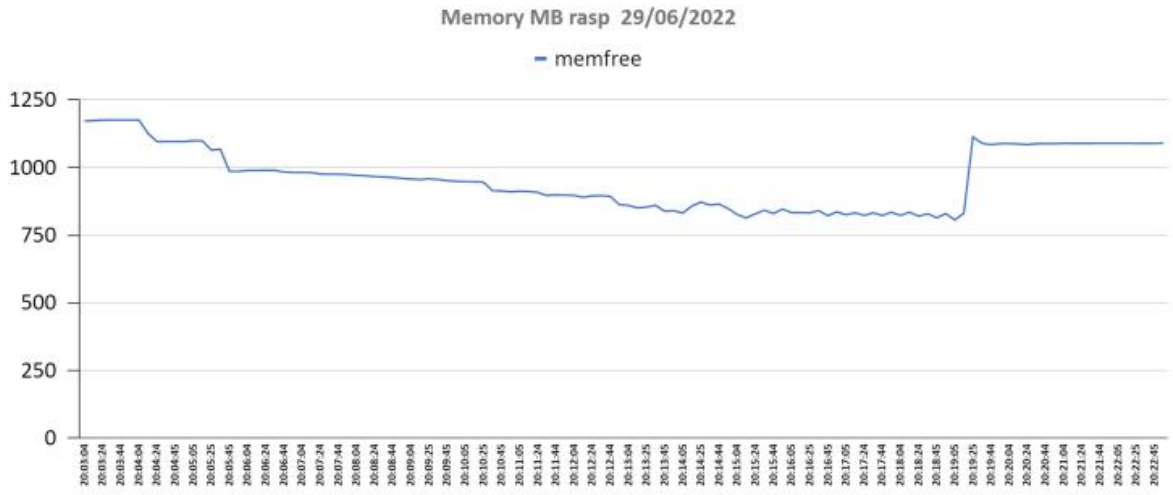
Figura 96 – Pacotes de rede x tempo na *Raspberry Pi*



Fonte: nmon analyser

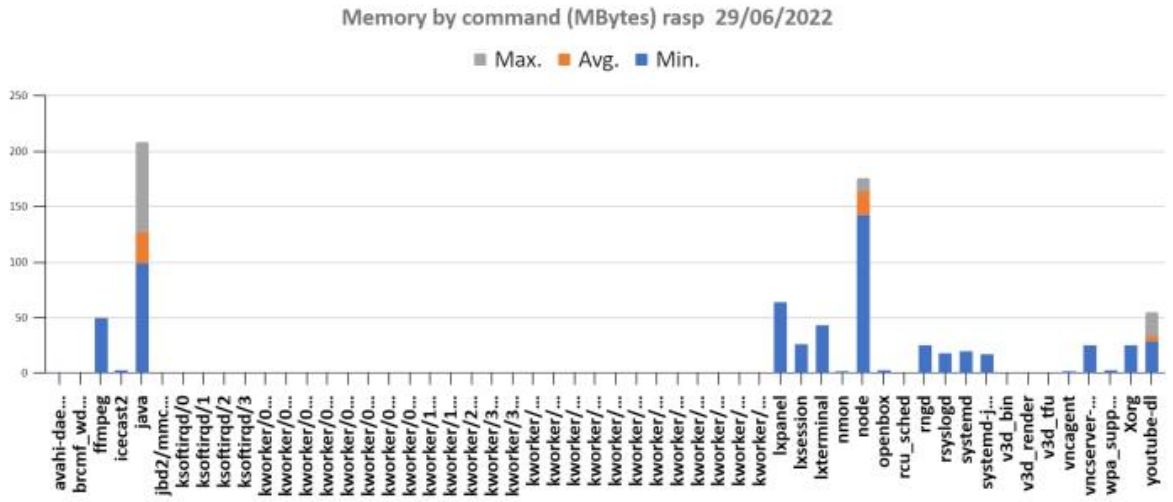
Consumo (ou disponibilidade) de memória:

Figura 97 – Memória livre x tempo na *Raspberry Pi*



Fonte: nmon analyser

Figura 98 – Memória utilizada x comando na *Raspberry Pi*



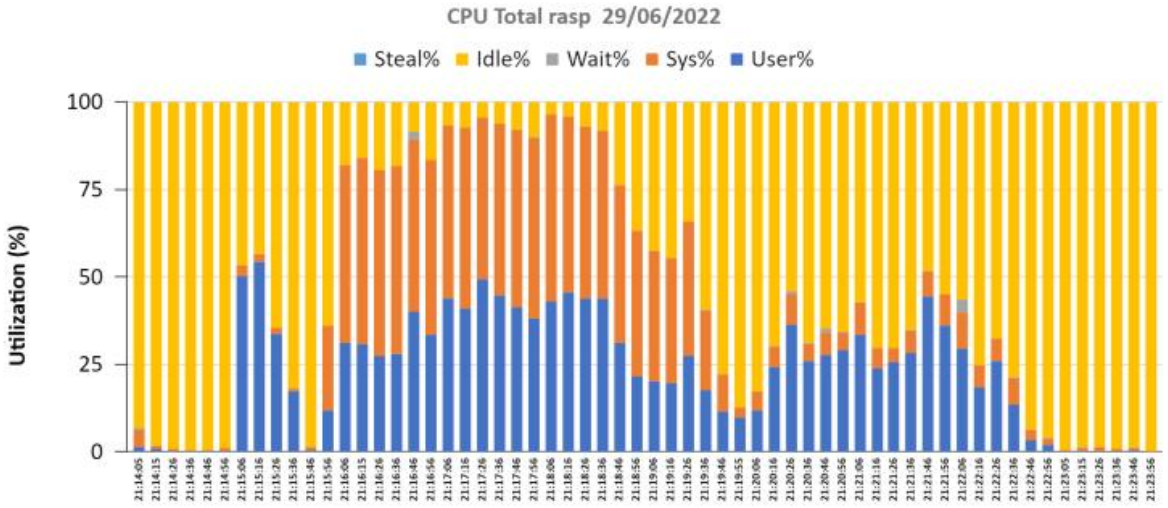
Fonte: nmon analyser

Cenário aleatório:

Neste cenário, as aplicações foram iniciadas e os clientes conectados, aproximadamente, entre 21:14 e 21:15. Pedidos de músicas aleatórios foram feitos até aproximadamente 21:22. Após esse momento, as aplicações foram finalizadas.

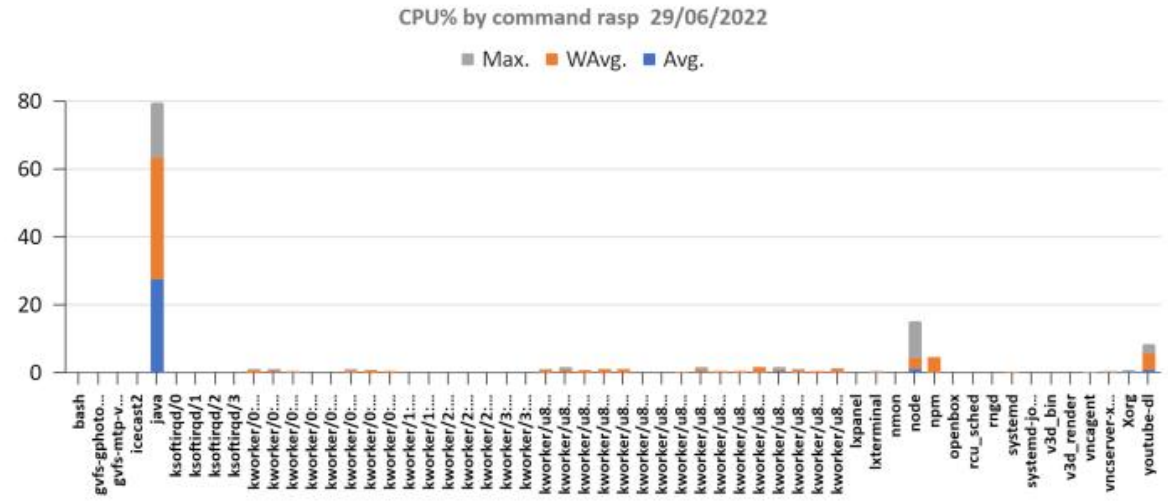
Consumo de CPU:

Figura 99 – Resumo de uso de sistema x tempo na *Raspberry Pi*



Fonte: nmon analyser

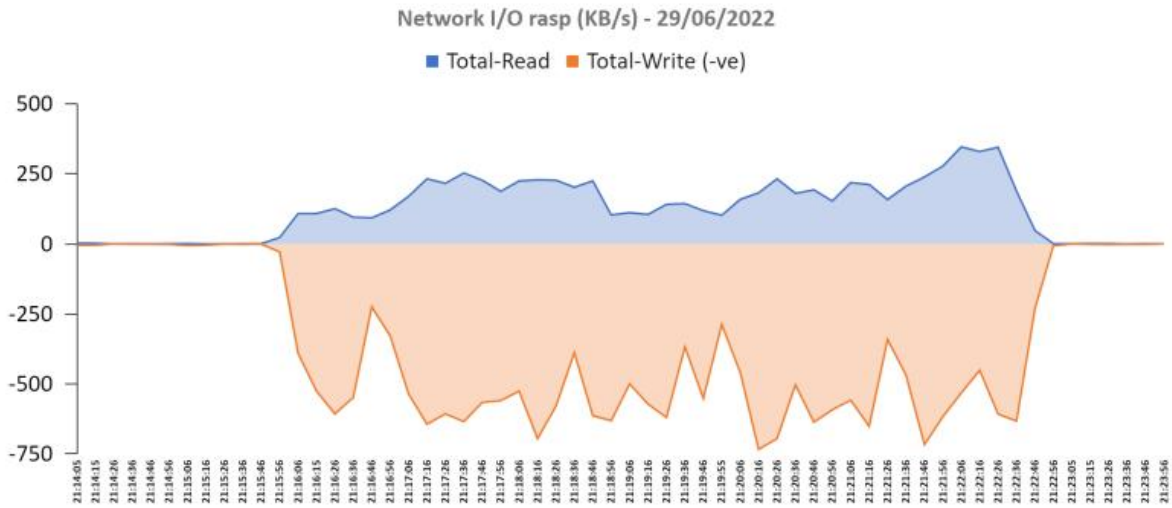
Figura 100 – Uso de CPU x comando na *Raspberry Pi*



Fonte: nmon analyser

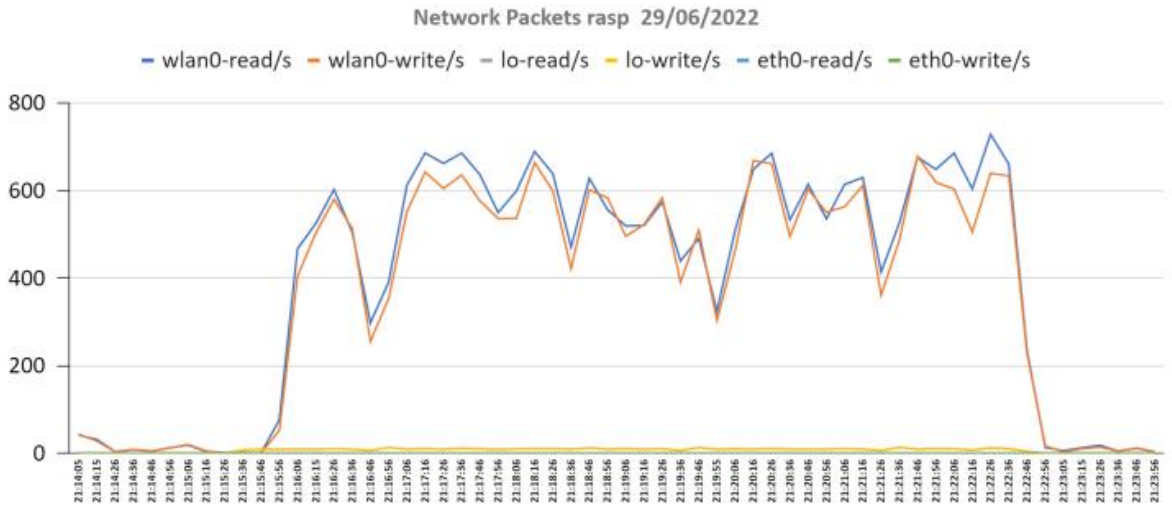
Consumo de rede:

Figura 101 – Consumo de rede x tempo na *Raspberry Pi*



Fonte: nmon analyser

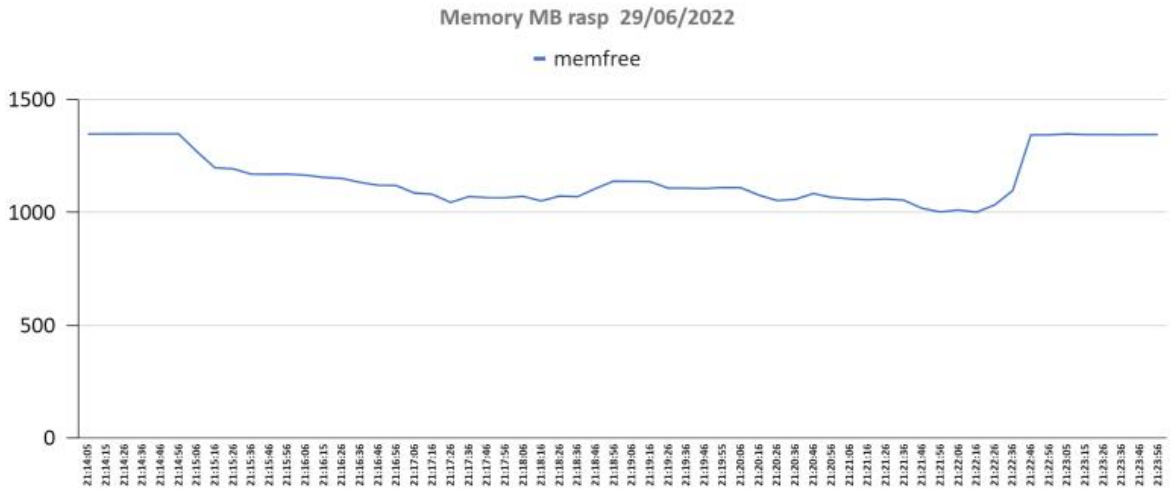
Figura 102 – Pacotes de rede x tempo na *Raspberry Pi*



Fonte: nmon analyser

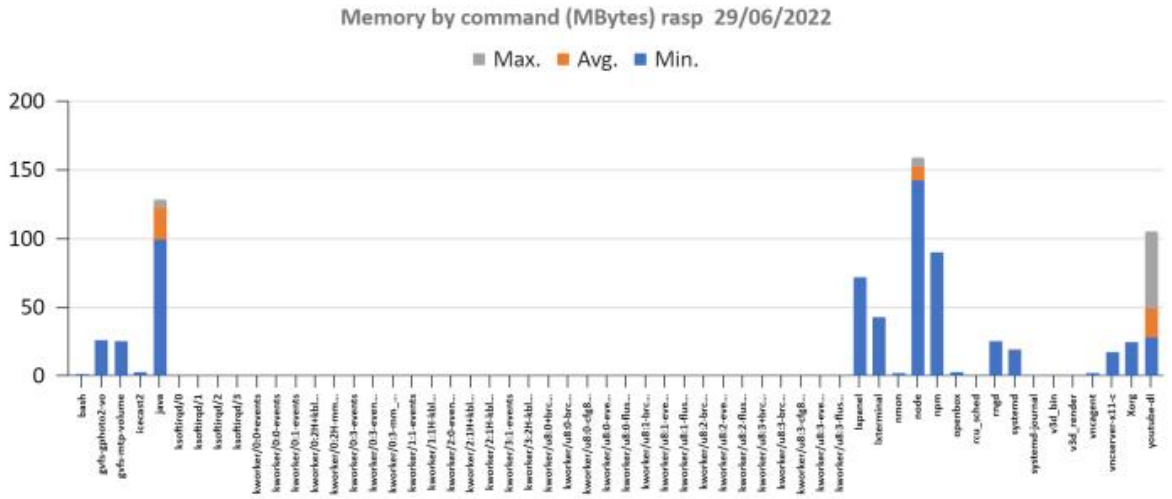
Consumo (ou disponibilidade) de memória:

Figura 103 – Memória disponível x tempo na *Raspberry Pi*



Fonte: nmon analyser

Figura 104 – Uso de memória x comando na *Raspberry Pi*



Fonte: nmon analyser

Os principais problemas e soluções podem ser visualizados no quadro 2.

Quadro 2 – Principais problemas e soluções encontrados no projeto

Problema	**Solução**	**Observação**
Dificuldade em reproduzir áudio em alta frequência (> 8KHz).	Melhoria no algoritmo de *buffer* circular no *Stream Server* e cliente ESP32.	Precisaria de uma melhoria ainda maior para alcançar valores melhores (> 16KHz).
Dificuldade em decodificar o áudio recebido do YouTube durante download para o formato necessário.	Algoritmo de decodificação *on-the-fly* no *Stream Server* usando *FFmpeg*.	Gera alguns *clicks* no início da reprodução. Melhoria no algoritmo poderia resolver esse problema.
Dificuldade em utilizar a tecnologia I2S.	Mais leitura das documentações disponíveis e pesquisa de exemplos online.	
Problema de travamento do *Stream Server* com múltiplas conexões.	Algoritmo de controle de uso de *sockets* resolvendo problema de *memory leak*.	
Problema de ruído encontrado na reprodução ao utilizar I2S (para frequências maiores).	Melhoria no algoritmo de *buffer* circular do cliente ESP32 diminuiu o problema, mas não o resolveu.	Não foi resolvido.
Interpretação ruim das variáveis dos comandos, por parte da *Wit AI*.	Treinamento.	
Dificuldade na interpretação dos comandos.	Resolução do conflito de arquivos por separação das gravações por nome (além da data em milissegundos).	Correção simples, mas a fonte do problema foi difícil de ser encontrada.
Confusão no recebimento e execução dos comandos. Execuções repetidas.	Melhoria no controle de comandos, com adição de *flags* de estado para sincronia.	

Fonte: Autoria própria (2022).

Em resumo, os problemas finais não solucionados são:

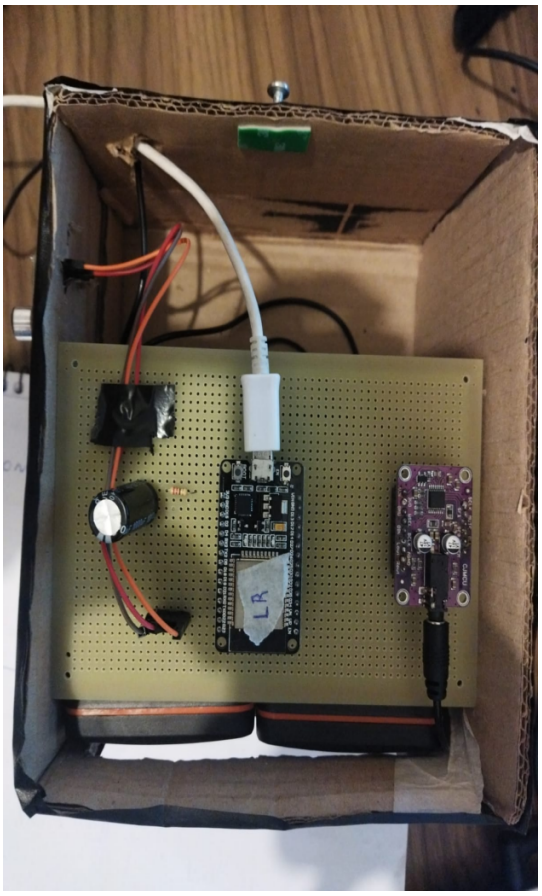
- Forte ruído com mais de uma conexão, após um certo período de tempo;

- Demora na reprodução do áudio, após interpretação, quando está rodando na *Raspberry*. Associada à falta de recursos;
- Dificuldade na interpretação, por parte da *Wit AI*, da música escolhida. Problema associado à diferença da língua;
- Baixa taxa de acerto de interpretação da entidade de música (aprox. 53%).

3.1 Fotos

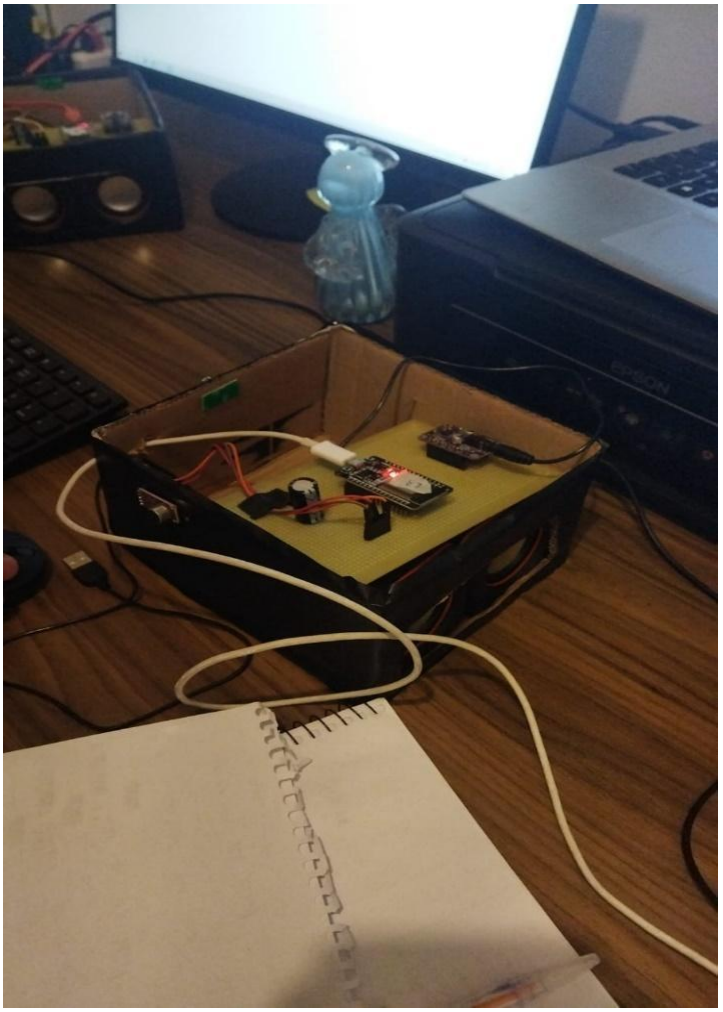
Os clientes do projeto com cada componente posicionado estão nas figuras 105 a 110.

Figura 105 – Sistema montado - superior



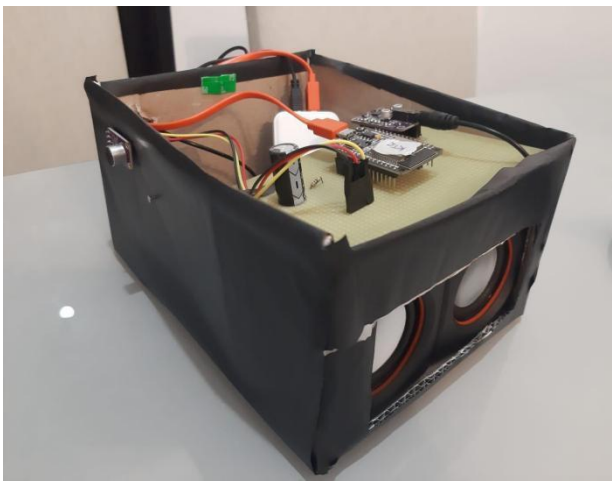
Fonte: Autoria própria (2022).

Figura 106 – Sistema montado - longe



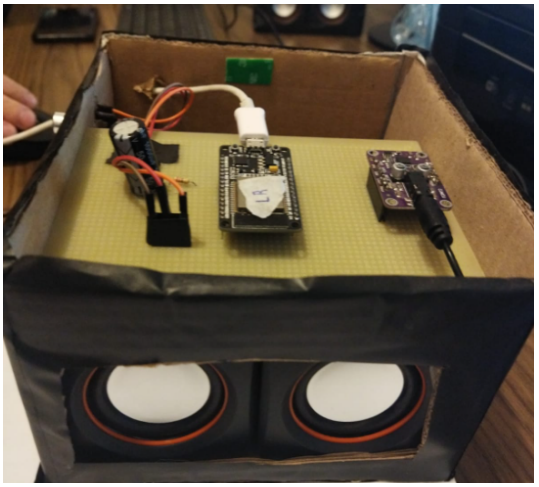
Fonte: Autoria própria (2022).

Figura 107 – Sistema montado - lateral



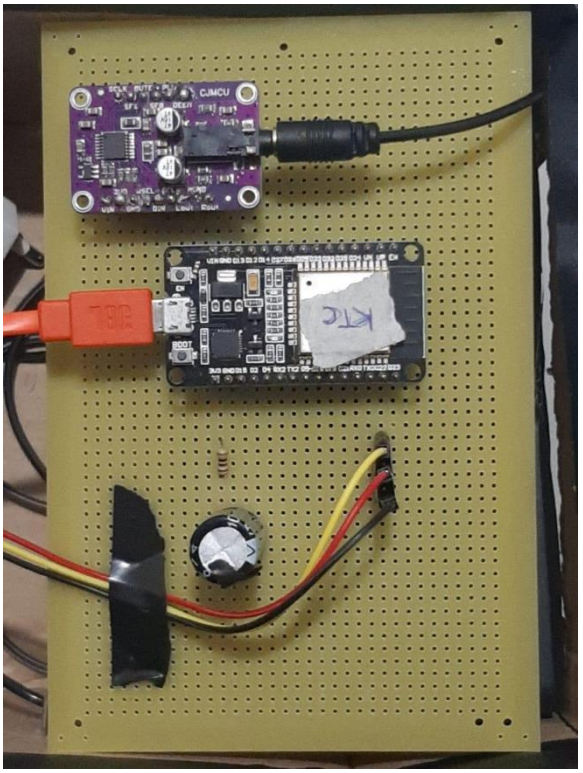
Fonte: Autoria própria (2022).

Figura 108 – Sistema montado - frontal



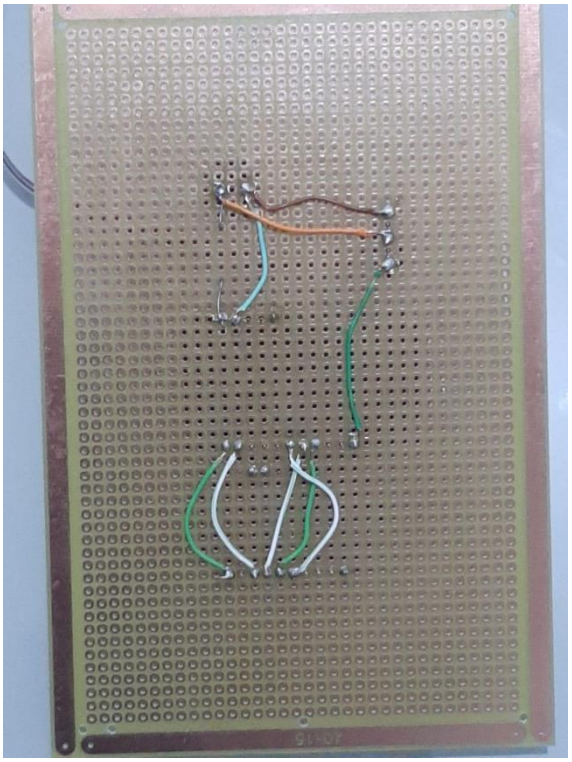
Fonte: Autoria própria (2022).

Figura 109 – Circuito montado - frente



Fonte: Autoria própria (2022).

Figura 110 – Circuito montado - verso



Fonte: Autoria própria (2022).

4 CONCLUSÃO

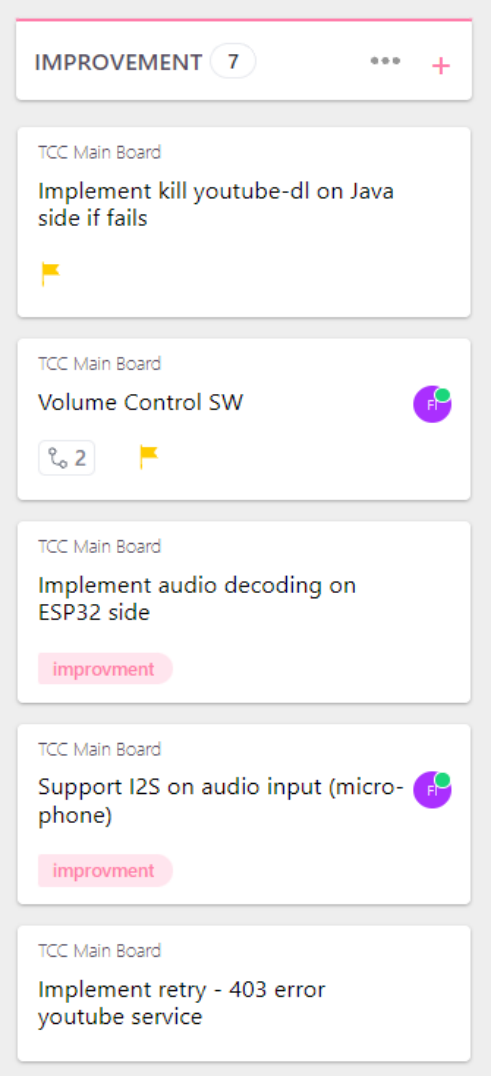
As tecnologias utilizadas para o desenvolvimento do projeto trouxeram abordagens que possibilitaram o uso de conhecimentos obtidos e técnicas praticadas durante a graduação de Engenharia Eletrônica.

Diante dos resultados, nota-se que os desafios encontrados proporcionaram uma reflexão além das teorias e práticas analisadas/executadas durante o desenvolvimento do projeto. Características do ambiente e do sistema, como hardware e capacidade da banda de internet, influenciaram diretamente nos resultados e validações. Ainda mais, o próprio alcance da internet provou-se muito influente no tempo de download das músicas, como também na comunicação entre os clientes e o servidor para o reconhecimento do comando de voz e a reprodução dos pacotes de áudio.

Alguns dos problemas encontrados foram amenizados (havia ainda espaço para melhoria) a partir da melhoria de algoritmos utilizados na solução do projeto, como a dificuldade de reprodução em alta frequência e a questão do chiado. Outros problemas foram solucionados com o estudo direcionado da teoria, como a dificuldade em entender e utilizar o protocolo I2S na entrada e saída de áudio, ou com a aplicação de uma nova abordagem, como utilizar o *FFmpeg* para decodificar o *stream* de áudio oriundo do *YouTube*, em tempo real, para o formato ideal do projeto.

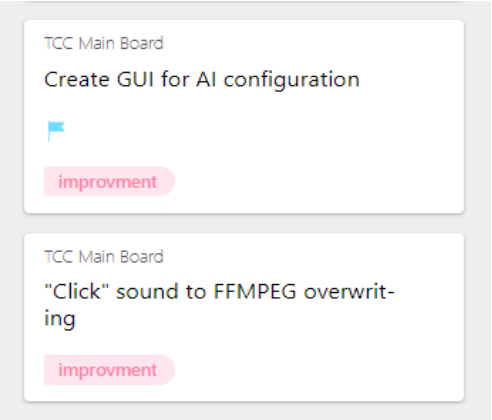
Além disso, existem alguns problemas secundários e melhorias que seriam interessantes, mas que foram deixadas de lado por ser um projeto acadêmico. Essas melhorias estão nas figuras 111 e 112 abaixo, dentro da *Kanban Board* utilizada no desenvolvimento do projeto.

Figura 111 – Tarefas criadas como melhorias na ferramenta do *Clickup 01*



Fonte: Click Up (2021, p. 1)

Figura 112 – Tarefas criadas como melhorias na ferramenta do *Clickup 02*



Fonte: Click Up (2021, p. 1)

Entre essas, apenas a tarefa "*Implement audio decoding on ESP32 side*" impacta severamente no projeto. Essa tarefa seria a de implementar um sistema de decodificação do áudio para *.wav* já no próprio cliente ESP32, aumentando a velocidade de transmissão, diminuindo o consumo de banda e, ainda mais importante, removendo por completo o ruído (evitando a necessidade de um *buffer* circular mais complexo).

As outras, em resumo, dizem respeito a:

- Finalizar processos "zumbi" (processos que não foram encerrados corretamente);
- Habilitar o controle de volume via comando de voz (este, apesar de não ser fundamental, seria bastante interessante);
- Alterar o uso do ADC interno da placa para uso do hardware dedicado para I2S, o que traria uma melhoria no sinal das gravações;
- Implementar uma forma de refazer a requisição que a ferramenta *YouTube DL* envia para o *YouTube* após o assistente de voz receber uma resposta HTTP *Forbidden* (código 403) do *Youtube* (não houve autorização para realizar a requisição)— o que ocorre ocasionalmente, logo, não foi considerado muito importante;
- Criar uma UI de configuração (não é tão importante, por ser um sistema de baixo tempo de configuração, mas seria interessante para que o cliente pudesse fazer alterações sem necessidade de suporte).
- Remover o som de "*click*" gerado pelo *FFmpeg* (ocorre apenas nos primeiros segundos de reprodução, logo, foi descartado como não importante).

A possibilidade de construir um sistema de áudio residencial controlado por voz foi dada como um grande desafio desde o início e, durante o desenvolvimento do projeto, vários empecilhos foram superados com muito aprendizado. Poder implementar esse sistema com dispositivos pequenos, relativamente baratos e fáceis de encontrar no mercado foi algo animador para a equipe, principalmente quando foi possível fazer os primeiros testes "ponta a ponta" funcionarem.

Durante o desenvolvimento do projeto, foram utilizados vários conhecimentos, técnicas e tecnologias, tanto na parte de hardware, como na parte de software. Entre eles, cita-se os principais abaixo:

- *Java;*
- C;
- *Typescript.*
- *Maven;*
- NPM;
- *Runtime Node.js;*
- Comunicação via *Socket;*
- REST API;
- Inteligência Artificial *Wit AI.*
- *YouTube DL;*
- *FFmpeg;*
- *Audacity;*
- Protocolo I2S;
- Hardwares de gravação e reprodução de áudio;
- Filtragem de sinal;
- Ferramentas de gerenciamento de projetos, como *Kanban Board;*
- Diagramas de tempo;
- Soldagem de componentes em placa perfurada;
- Sistema operacional “*Unix based*”;
- Sistema *multi-thread* e sinalização por semáforos;
- *Framework Spring Boot* (SPRING, 2021)*;*
- Algoritmos leves de transmissão e consumo de dados.

REFERÊNCIAS

ARDUINO. **Arduino - Home.** 2022. Disponível em: https://www.arduino.cc/en/software/. Acesso em: 7 de Fevereiro de 2020.

Audacity ®. **Audacity Home**. 2022. Disponível em: https://www.audacityteam.org/. Acesso em: 22 de Agosto de 2020.

BAELDUNG. **Spring MVC streaming and SSE request processing**. 2022. Disponível em: https://www.baeldung.com/spring-mvc-sse-streams. Acesso em: 13 de Abril de 2020.

BOLTON. D. **youtube-dl/README.md**. 2022. Disponível em: https://github.com/ytdl-org/youtube-dl/blob/master/README.md#readme. Acesso em: 2 de Julho de 2020.

BOTPRESS. **Chatbot for developers**. 2022. Disponível em: https://botpress.com/. Acesso em: 25 de Julho de 2021.

CANONICAL LTD. **Ubuntu**. 2022.Disponível em: https://ubuntu.com/. Acesso em: 15 de Junho de 2021.

CHRIS. **ESP32 I2S audio**. 2021. Disponível em: https://github.com/atomic14/esp32_audio. Acesso em: 15 de Julho de 2021.

CHOUDHARY. A. **DIY ESP32 based audio player**. 2020. Disponível em: https://circuitdigest.com/microcontroller-projects/esp32-based-audio-player. Acesso em: 22 de Abril de 2020.

CLICK UP HELP CENTER. **How can we help?**. 2022. Disponível em: https://docs.clickup.com/en/. Acesso em: 10 de Junho de 2021.

CNX SOFTWARE. **ESP8266 AND ESP32 differences in one single table**. 2016. Disponível em: https://www.cnx-software.com/2016/03/25/esp8266-and-esp32-differences-in-one-single-table/. Acesso em: 11 de Julho de 2021.

DAVIS. I. **Installing Maven on Raspberry Pi 3 OS Raspbian**. 2017. Disponível em: https://stackoverflow.com/questions/44622336/installing-maven-on-raspberry-pi-3-os-raspbian. Acesso em: 9 de Setembro de 2020.

DIAZ, CARLOS M. E. O. A. **jetson nano i2s CJMCU-1334**. 2020. Disponível em: https://www.youtube.com/watch?v=6UWz2LeeAks&ab_channel=CarlosMaximilianoEstebaOlmosdeAguileraD%C3%ADaz. Acesso em: 15 de Junho de 2021.

DISCORD. **Discord | Your place to talk and hang out**. 2022. Disponível em: https://discord.com/. Acesso em: 7 de Fevereiro de 2020.

ELECTRORULES. **ESP32 pinout reference: What GPIO pins do you use?**. 2021. Disponível em: https://www.electrorules.com/esp32-pinout-reference/. Acesso em: 12 de Junho de 2020.

ESPRESSIF. **API reference**. 2022. Disponível em: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/index.html. Acesso em: 21 de Julho de 2021.

ESPRESSIF SYSTEMS. **Audio development framework documentation**. 2019. Disponível em: https://docs.espressif.com/projects/esp-adf/en/latest/api-reference/streams/index.html#http-stream. Acesso em: 12 de Abril de 2020.

ESPRESSIF SYSTEMS. **FreeRTOS overview**. 2022. Disponível em: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/freertos.html. Acesso em: 9 de Maio de 2021.

ESPRESSIF SYSTEMS. **Wi-Fi & Bluetooth MCUs and AIoT solutions**. 2022. Disponível em: https://www.espressif.com/en/products/socs. Acesso em: 7 de Fevereiro de 2020.

FFMPEG. **Documentation**. 2022. Disponível em: https://ffmpeg.org/documentation.html. Acesso em: 25 de Julho de 2021.

GEEKSFORGEEKS. **Program for round robin scheduling**. 2022. Disponível em: https://www.geeksforgeeks.org/program-round-robin-scheduling-set-1. Acesso em: 7 de Fevereiro de 2020.

GIT. **Git**. 2022. Disponível em: https://git-scm.com/. Acesso em: 12 de Abril de 2020.
GIT HUB. **GitHub documentation**. 2022. Disponível em: https://docs.github.com/pt. Acesso em: 15 de Junho de 2021.

GOOGLE. **Google meet**. 2022. Disponível em: https://meet.google.com/. Acesso em: 5 de Maio de 2021.

GOOGLE. **gson**. 2022. Disponível em: https://github.com/google/gson. Acesso em: 11 de Abril de 2021.

GOOGLE CLOUD. **Dialogflow documentation**. 2022. Disponível em: https://cloud.google.com/dialogflow/docs. Acesso em: 15 de Junho de 2021.

GOOGLE DEVELOPERS. **API reference**. 2019. Disponível em: https://developers.google.com/youtube/v3/docs?hl=pt_br. Acesso em: 5 de Maio de 2021.

GOOGLE IDENTITY PLATFORM. **Como usar OAuth 2.0 para acessar as APIs do Google**. 2022. Disponível em: https://developers.google.com/identity/protocols/oauth2. Acesso em: 11 de Agosto de 2021.

HOGAN, B. **Como instalar o Java com o Apt no Ubuntu 20.04**. 2020. Disponível em: https://www.digitalocean.com/community/tutorials/how-to-install-java-with-apt-on-ubuntu-20-04-pt. Acesso em: 9 de Setembro de 2020.

IBM - **Deutschland (n.d.). IBM Watson**. 2022. Disponível em: https://www.ibm.com/watson. Acesso em: 2 de Fevereiro de 2020.

JETBRAINS. **IntelliJ IDEA: The capable & ergonomic Java IDE by JetBrains**. 2022. Disponível em: https://www.jetbrains.com/idea/. Acesso em: 25 de Julho de 2021.

JULIAN. **Stream your audio on the ESP32**. 2018. Disponível em: https://www.hackster.io/julianfschroeter/stream-your-audio-on-the-esp32-2e4661. Acesso em: 5 de Maio de 2021.

LINDEVS. **Install youtube-dl on Raspberry Pi**. 2021. Disponível em: https://lindevs.com/install-youtube-dl-on-raspberry-pi/. Acesso em: 10 de Setembro de 2020.

LINUXIZE. **How to install and Use FFmpeg on Ubuntu 20.04**. 2020. Disponível em: https://linuxize.com/post/how-to-install-ffmpeg-on-ubuntu-20-04/. Acesso em: 9 de Setembro de 2020.

LINUX.ORG. **Linux.org. 2022**. Disponível em: https://www.linux.org/. Acesso em: 12 de Abril de 2020.

MYCROFT. **The Open Source privacy-focused voice assistant**. 2022. Disponível em: https://mycroft.ai/. Acesso em: 7 de Fevereiro de 2020.

MICROSOFT. **TypeScript: JavaScript with syntax for types**. 2022. Disponível em: https://www.typescriptlang.org/. Acesso em: 15 de Junho de 2021.

MICROSOFT. **Visual Studio Code - Code editing. Redefined**. 2021. Disponível em: https://code.visualstudio.com/. Acesso em: 5 de Maio de 2021.

NPM. **axios-npm**. 2022. Disponível em: https://www.npmjs.com/package/axios. Acesso em: 5 de Abril de 2021.

NPM. **nodemon**. 2019. Disponível em: https://www.npmjs.com/package/nodemon. Acesso em: 1 de Abril de 2021.

npm. **npm**. 2022. Disponível em: https://www.npmjs.com/. Acesso em: 12 de Abril de 2020.

NPM. **rxjs-npm**. 2022. Disponível em: https://www.npmjs.com/package/rxjs. Acesso em: 5 de Abril de 2021.

NPM. **youtube-search**. 2020. Disponível em: https://www.npmjs.com/package/youtube-search. Acesso em: 1 de Abril de 2021.

OLEGKUNITSYN. **Libshout binding for streaming audio to Icecast servers from Java applications**. 2022. Disponível em: https://github.com/OlegKunitsyn/libshout-java. Acesso em: 5 de Maio de 2021.

OOKLA. **Speed Test**. 2022. Disponível em: https://www.speedtest.net/pt. Acesso em: 15 de Junho de 2021.

OPENJS FOUNDATION. **Node.js**. 2022. Disponível em: https://nodejs.org/. Acesso em: 25 de Julho de 2021.

OPUS. **Opus codec**. 2020. Disponível em: https://opus-codec.org/. Acesso em: 7 de Fevereiro de 2020.

ORACLE. **Document** (Java Platform SE 8). 2022. Disponível em: https://docs.oracle.com/javase/8/docs/api/org/w3c/dom/Document.html. Acesso em: 15 de Junho de 2021.

ORACLE. **ProcessBuilder** (Java Platform SE 7). 2020. Disponível em: https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html. Acesso em: 5 de Maio de 2021.

PYTHON.ORG. **Welcome to Python.org**. 2022. Disponível em: https://www.python.org/. Acesso em: 7 de Fevereiro de 2020.

PLURALSIGHT. **JavaScript**. 2022. Disponível em: https://www.javascript.com/. Acesso em: 7 de Fevereiro de 2020.

RASPBERRY PI. **Raspberry Pi documentation**. 2022. Disponível em: https://www.raspberrypi.com/documentation/. Acesso em: 20 de Abril de 2021.

SANTOS. D. M. dos. **Conversor analógico-digital**. 2022. Disponível em: https://www.infoescola.com/eletronica/conversor-analogico-digital/. Acesso em: 24 de Julho de 2020.

SLUIJS. T. V. D. **Python 3.9.5 on Raspberry Pi installation step by step**. 2021. Disponível em: https://itheo.tech/install-python-395-on-raspberry-pi-step-by-step. Acesso em: 12 de Agosto de 2022.

S. P. I, Inc. **Debian | The universal operating system**. 2022. Disponível em: https://www.debian.org/. Acesso em: 7 de Fevereiro de 2020.

SOUZA. D.R. **danrezsendes** / tcc. 2022. Disponível em: https://github.com/danrezsendes/tcc. Acesso em: 7 de Fevereiro de 2020.
SPRING. **Spring Boot**. 2022. Disponível em: https://spring.io/projects/spring-boot. Acesso em: 11 de Abril de 2021.

THE APACHE SOFTWARE FOUNDATION. **Welcome to Apache Maven**. 2022. Disponível em: https://maven.apache.org/. Acesso em: 12 de Abril de 2020.

THE ECLIPSE FOUNDATION. **Eclipse IDE for Java developers**. 2022. Disponível em: https://www.eclipse.org/downloads/packages/release/oxygen/3a/eclipse-ide-java-developers. Acesso em: 12 de Abril de 2020.

TUTORIALS, R. N. **ESP32 HTTP GET and HTTP POST with Arduino IDE**. 2022. Disponível em: https://randomnerdtutorials.com/esp32-http-get-post-arduino/. Acesso em: 13 de Abril de 2020.

WHATSAPP LLC. **WhatsApp.com**. 2022. Disponível em: https://www.whatsapp.com/. Acesso em: 7 de Fevereiro de 2020.

WIT AI. **Getting started with Wit.AI**. 2022. Disponível em: https://wit.ai/docs. Acesso em: 21 de Julho de 2021.

XIANIC BLOG. **Installing maven on the Raspberry Pi**. 2015. Disponível em: https://xianic.net/2015/02/21/installing-maven-on-the-raspberry-pi/. Acesso em: 9 de Setembro de 2020.

YOUTUBE DATA API. **Search list**. 2019. Disponível em: https://developers.google.com/youtube/v3/docs/search/list?hl=pt_br. Acesso em: 10 de Agosto de 2021.