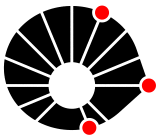
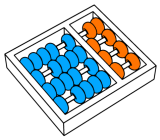




Monitoramento de colmeias com Internet das Coisas

M. C. Rosa *L. C. Castello* *C. A. A. Trujillo*
L. F. Bittencourt

Relatório Técnico - IC-PFG-23-28
Projeto Final de Graduação
2023 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Monitoramento de colmeias com Internet das Coisas

Marcos C. Rosa* Lucas C. Castello† Carlos A. A. Trujillo‡
Luiz F. Bittencourt§

Resumo

Este é o relatório do projeto realizado como Trabalho de Conclusão de Curso do Instituto de Computação, em parceria com o Prof. Roberto Greco do Instituto de Geociências, cujo objetivo é desenvolver um sistema para coleta de dados de colmeias que serão instalados em escolas.

O sistema permite receber os dados da colmeia via Wi-Fi em um dispositivo móvel, mantendo-se numa distância segura das abelhas. Para este projeto, foi fornecida uma placa desenvolvida pelo Prof. Fabiano Fruett com sensores de temperatura, umidade, som e proximidade. A programação do software da placa para a coleta e transmissão de dados, bem como a implementação de um aplicativo Android para visualização dos resultados são os objetos de estudo deste relatório.

1 Introdução

No ano de 2022, o professor Roberto Greco do Instituto de Geografia da UNICAMP começou um projeto para a instalação de colmeias no assentamento Milton Santos, em Americana SP. Como parte deste projeto, alguns alunos do Instituto de Computação da UNICAMP, orientados pelo professor Luiz Fernando Bittencourt, criaram uma ferramenta para monitorar as colmeias, como trabalho de conclusão de curso[44]. Tal ferramenta consistia em uma placa com um microcontrolador ESP32 que lia sensores de temperatura, umidade e som e enviava os valores por Bluetooth a um aplicativo desenvolvido por eles.

Este projeto apresentou alguns erros pouco após sua implementação, levando o prof. Roberto a procurar a ajuda do professor Fabiano Fruett da Faculdade de Engenharia Elétrica e Computação, o qual desenvolveu uma nova versão da placa. Esta conta com uma Raspberry Pi Pico W, que possui um microprocessador RP2040 com 2MB de memória flash, possibilidade de conexão WiFi e Bluetooth, além de alguns sensores de temperatura, umidade, pressão, proximidade e som.

*m240815@dac.unicamp.br
†l172641@dac.unicamp.br
‡carlosat@unicamp.br
§bit@ic.unicamp.br

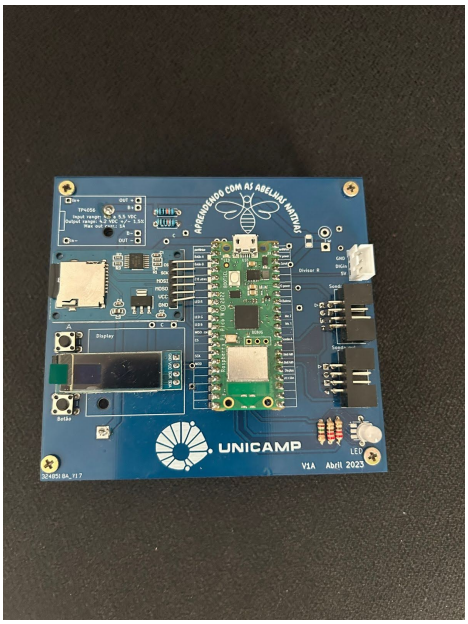


Figura 1: Placa elaborada para monitoramento de colméias

O relatório a seguir relata a programação da placa confeccionada, que desta vez terá como destino escolas com as colméias implantadas, bem como de um novo aplicativo para a comunicação com o microcontrolador, baseados no conceito de Internet das Coisas e no projeto do ano anterior.

A seguir estão as seções: Objetivo, com os objetivos do projeto; Metodologia, com a forma com a qual o projeto foi estruturado e elaborado;Comunicação, mostrando a forma como se comunicam o microcontrolador e o aplicativo; Microcontrolador, com a forma que foi implementada as funcionalidades da placa; Aplicativo, com a forma que foi implementado o aplicativo mobile para requisição e leitura dos dados; Documentação, apresentando os links dos repositórios originados deste projeto; Resultados, descrevendo o desfecho e Modo de uso, relatando a forma de usar o projeto; Conclusão, com as considerações finais; e Referências Bibliográficas, com as fontes usadas pelos autores.

2 Objetivos

Dadas as necessidades levantadas pelo prof. Roberto e a placa providenciada pelo prof. Fabiano, temos como objetivo principal:

- Fazer com que um sensor (temperatura, umidade, som e movimentação de abelhas) instalado em uma colmeia se comunique via Wi-Fi com um aplicativo Android capaz de apresentar dados recebidos através de gráficos.

Para alcançar este objetivo, o dividimos em pontos menores que serão exemplificados no decorrer do relatório:

- Coleta de dados de temperatura, umidade, som e de movimentação nas entradas da colméia
- Salvar os dados coletados em um cartão microSD
- Conectar o microcontrolador ao WiFi local usando Bluetooth Low Energy
- Enviar os dados para o aplicativo via WiFi
- Mostrar os dados em forma de gráficos para cada métrica ou agrupados
- Mostrar as mínimas e máximas de cada métrica

3 Metodologia

Dadas as áreas de expertise dos autores e a forma como o projeto naturalmente se divide em aplicativo e microcontrolador, o aluno Lucas Castello ficou responsável por este, enquanto o aluno Marcos Rosa ficou responsável por aquele. Reuniões semanais foram estabelecidas para discussão da interface entre as duas partes, além de relatar problemas, levantar possíveis soluções e ocasionalmente chamar uma pessoa de fora para ajudar. Nessas reuniões, ficou estabelecido o seguinte esquema para o projeto:

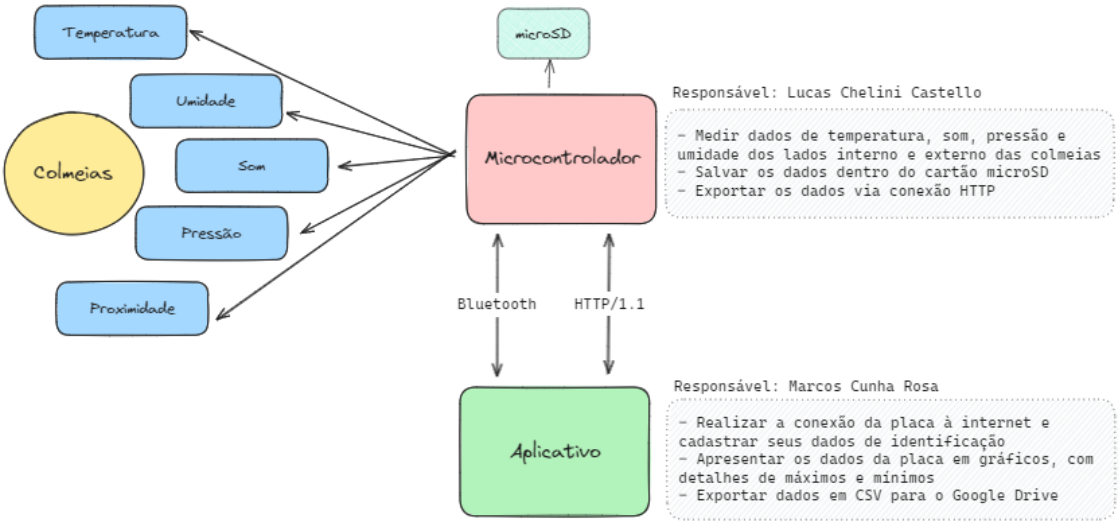


Figura 2: Esboço da arquitetura do projeto como um todo

4 Comunicação

Uma vez estabelecida a divisão de trabalho, o próximo passo foi definir a comunicação entre o aplicativo e a placa, que seria o ponto comum a partir do qual ambas as frentes pautaram seu

desenvolvimento. O ponto principal desta comunicação seria a possibilidade de o aplicativo fazer requisições à placa, e esta devolver a devida resposta.

O projeto do ano anterior utilizava Bluetooth para o envio das medidas feitas. Essa decisão foi tomada por falta de Wi-Fi no assentamento em que seriam inseridas as placas. Como agora o destino das placas são escolas que possuem Wi-Fi, essa limitação não existe mais, e por isso esta foi escolhida como o meio de comunicação entre as partes. Para realizar a comunicação através de um protocolo amplamente utilizado, foi escolhido o HTTP/1.1, apresentando como requisições principais a GET e POST, e retornando respostas através de códigos de estado (*status code*)[35].

Além disso, para a placa conectar-se a primeira vez à rede local, foi utilizado um mecanismo de comunicação Bluetooth entre as partes, que será abordado em mais detalhes nas seções Placa e Aplicativo.

5 Placa

Como dito, a implementação do hardware da placa foi feita pelo professor Fabiano Fruett da Faculdade de Engenharia Elétrica e Computação (FEEC) da UNICAMP, anteriormente à entrada dos autores no projeto. A seguir estão listados os componentes da placa e um resumo de suas funções:

- Raspberry Pi Pico W, que por sua vez é composta por um microcontrolador RP2040 com 2MB de memória flash, suporte para conexão Wifi, Bluetooth e Bluetooth Low Energy, entrada micro USB B e mais 40 pinos que desempenham diferentes funções.
- Sensor de proximidade por Ultrassom E18-D80NK - NPN, para detecção de saída e entrada de abelhas da colméia.
- 2 microfones de eletreto Módulo MAX4466, para medição do som interno e externo da colméia.
- 2 sensores BME680, para leitura de dados climáticos como temperatura, umidade e pressão, tanto para a parte interna da colméia quanto para a parte externa.
- Display LCD que não foi utilizado
- Luz LED que não foi utilizada
- 2 push button 4 terminais que não foram usados.

5.1 Objetivos da placa

Os objetivos da placa podem ser divididos em três partes referentes aos dados, conforme a seguir:

1. Leitura dos sensores: Implementar um método que realize uma leitura dos sensores conectados à placa a cada intervalo de tempo definido pelo usuário do sistema.

2. Armazenamento das medições: Gravar no microSD os dados lidos dos sensores, juntamente com o horário da leitura.
3. Ouvir por conexões: Responder a requisições de outros sistemas, retornando e salvando dados por meio da comunicação sem fio.

5.2 Arquitetura

O projeto do software da placa se iniciou com a definição da linguagem Micropython para a programação. Essa linguagem é uma implementação de Python 3 para microcontroladores, escondendo funcionalidades de baixo nível com a sintaxe de alto nível do Python. Tal escolha foi feita pela familiaridade do responsável com a linguagem Python tradicional, que utilizou o paradigma da Programação Orientada a Objetos e uma arquitetura baseadas em classes gerenciadoras para dividir as três funcionalidades em quatro classes: SensorsManager para a leitura; DatabaseManager para o armazenamento; e ServerManager para a comunicação via internet, e ConnectionsManager para gerenciamento da conexão da placa à internet ou bluetooth. A Figura 3 a seguir mostra uma esquematização das classes e suas relações.

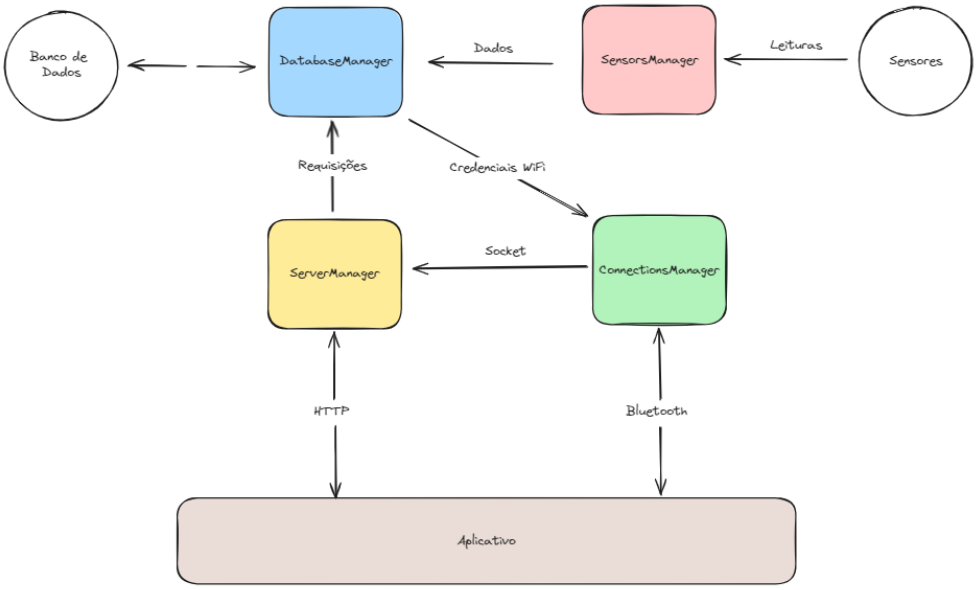


Figura 3: Diagrama representando as classes gerenciadoras da placa

5.3 Biblotecas externas

Além das bibliotecas padrão do Micropython, quatro foram incluídas no projeto, sob a pasta *lib*:

- *ble_simple_peripheral*, para lidar com o Bluetooth Low Energy, retirada do repositório do próprio Micropython[2].

- *ble_ advertising*, com o mesmo propósito e mesma origem da biblioteca *ble_ simple_ peripheral*.
- *sdcard*, para leitura e escrita em um cartão Micro SD[46].
- *bme680*, para leitura deste sensor. Esta é um port para Micropython de outra biblioteca desenvolvida pela Adafruit para sua linguagem Circuit Python[39] .

Todas as outras bibliotecas citadas nesta seção são padrão da linguagem Micropython

5.4 Estrutura do Software

Como dito, a estrutura se baseia na arquitetura Classes Gerenciadoras, na qual cada classe cuida de uma funcionalidade da placa. Cada classe é instanciada no arquivo main.py, que é rodado pela Raspberry Pi por padrão assim que a placa é conectada à energia.

O arquivo main também é o responsável pela inicialização das threads, que são responsáveis pela escuta de requisições e pela leitura dos sensores. Como ambas as atividades devem ser contínuas, uma única thread não seria capaz de lidar com o projeto. Assim, utilizou-se do fato de o microcontrolador da Raspberry Pi possuir dois núcleos, podendo rodar exatas duas threads.

5.4.1 Leitura

Para fazer a leitura dos sensores, a classe SensorManager precisa primeiramente instanciá-los, e isto por sua vez precisa dos pinos em que cada sensor está conectado à Raspberry Pi. Assim, segue a pinagem de cada sensor, bem como seus valores na interface de entrada e saída do software (GPIO).

- Proximidade: pino 4, GPIO 2;
- BME680 1: pinos 26 e 27, GPIO 20 e 21
- BME680 2: pinos 24 e 25, GPIO 24 e 25
- Microfone 1: pino 31, GPIO 26
- Microfone 2: pino 32, GPIO 27

Cada instância de sensor é salva nos atributos do gerenciador, para que uma nova não seja gerada a cada ciclo de leituras.

Este ciclo, como dito nos objetivos, deve ter duração estabelecida pelo usuário, ou seja, a cada X segundos a placa deve ter lido todos os sensores e salvado seus valores, com X definido pelo aplicativo. Na sub-seção 5.4.3 Resposta a Requisições será visto como este parâmetro é recebido, mas o que importa no momento é que é armazenado no atributo timer. Como o único sensor que precisa ser lido continuamente é o de proximidade, o timer atua encerrando o loop de leitura deste quando o tempo decorrido do início se torna maior que este. Encerrado o tempo, são chamadas as leituras dos outros sensores e, por fim, o gerenciador do banco de dados é chamado para salvar as leituras feitas neste ciclo. À exceção do sensor de proximidade, todos os sensores são lidos apenas uma vez por ciclo.

Quanto a este sensor, mostrado na Figura 4 abaixo, sua leitura é feita a cada 0.05 segundos, já que uma abelha passando a toda velocidade deve ser detectada, mas uma passando em velocidades menores não podem contar mais de uma vez. O sensor retorna 0 quando há algo em sua linha de detecção e 1 caso contrário, valores que são invertidos e somados a um atributo chamado *proximity_ counter*, que conta quantas abelhas passaram pelo sensor diariamente. A cada leitura, a função *localtime* da biblioteca padrão time é chamada para obtenção do dia do mês atual e comparada ao atribuído *current_ day*. Se forem diferentes, o loop de leitura de proximidade é encerrado independente do timer, para que o último valor do contador diário seja salvo e então zerado. O dia obtido pela função *localtime* é então atribuído ao *current_ day*.

Figura 4: Sensor de movimento
Fonte: https://www.usinainfo.com.br/sensor-de-proximidade/sensor-de-proximidade-e18-d80nk-infravermelho-npn-deteccao-3-a-80cm-2791.html

Quanto ao sensor sonoro, mostrado na Figura 5, sua leitura foi inicialmente feita utilizando a biblioteca ADC, que converte o valor analógico do sensor em valores digitais entre 0 e 65535. Como a capacidade máxima de leitura do sensor é 60dB, a equação para obter o valor lido em decibéis ficou:

```
Eq 1:  valor_real = 60 * valor_lido / 65535
```

Porém a Figura 6a mostra a leitura dos sensores com um deles sendo assoprado e então tendo uma voz falada bem próxima. Como é possível ver, não há diferença significativa entre ambos e os valores parecem aleatórios. Como forma de comparação, a Figura 6b mostra a mesma leitura sem os sensores conectados.

Um teste realizado foi a calibração dos sensores, que possuem um parafuso em sua traseira para isso. Um dos sensores teve seu parafuso girado ao máximo no sentido horário, enquanto o outro no anti-horário e o teste foi repetido, obtendo-se o mesmo resultado. Porém, valores diferentes foram obtidos quando o dedo do experimentador acidentalmente pressionou a parte de trás do sensor. Neste caso, a leitura feita ou beirava o zero, ou possuía valor máximo, não importando também o som emitido. Dessa forma, entende-se que de fato havia algo sendo lido, ainda que não o som desejado.

No tempo de realização deste relatório, não foi possível fazer o sensor de som funcionar apropriadamente. Há poucas fontes online para a leitura do MAX4466 usando Micropython, o que dificultou o trabalho e o tornou muitas vezes experimental. O professor Fabiano sugeriu a utilização de um sistema operacional em tempo real chamado NuttX, mas que ainda não foi testado e fica como sugestão de melhoria para próximas tentativas.

Figura 5: Sensor sonoro
Fonte: https://www.smartkits.com.br/modulo-microfone-c-ganho-ajustavel-gy-max4466

```
internal sound: 432
external sound: 416

internal sound: 37721
external sound: 36120

internal sound: 448
external sound: 432

internal sound: 34232
external sound: 34072

internal sound: 33496
external sound: 32904

internal sound: 35096
external sound: 33496

internal sound: 30935
external sound: 30951
```

(a) Sensores conectados

```
internal sound: 12899
external sound: 13331

internal sound: 13123
external sound: 13475

internal sound: 13971
external sound: 13923

internal sound: 14227
external sound: 14259

internal sound: 14275
external sound: 14339

internal sound: 13539
external sound: 13651

internal sound: 12403
external sound: 12675
```

(b) Sensores desconectados

Figura 6: Leitura dos sensores sonoros conectados(a) e desconectados(b)

Por fim, quanto ao BME680, mostrado na Figura 7 observa-se que são necessários dois pinos para sua leitura. Isso ocorre devido ao uso do barramento I2C, que precisa de uma entrada para dados e outra apenas para o clock.

Três bibliotecas para a leitura deste foram encontradas, porém apenas a referida na seção 5.3 retornou valores de fato. Como dito, esta é um port da biblioteca feita pela Adafruit para Circuit Python, criada em 2020 e aparentemente mantida apenas por uma pessoa. Como tal, é esperado que seja um trabalho em desenvolvimento e não tenha todas as funcionalidades perfeitamente implementadas.

De fato, não só acontece frequentemente de a instanciação da classe de leitura do BME680 resultar em erro, mas também compromete as outras funções da placa. Quando esta classe está instanciada, a ativação do Bluetooth resulta em timeout várias vezes, bem como a conexão WiFi. A leitura do banco de dados também fica desacelerada de tal forma que se torna impraticável sua realização. Por isso, foi tomada a decisão de deixar seu uso de lado até o momento.

Duas possibilidades de resolução foram levantadas: modificar a biblioteca de leitura que funcionou, ou trocar o tipo do sensor. A primeira se baseia na possibilidade de haver vazamentos de memória na instanciação, algo que foi experimentado algumas vezes em outros

pontos do código durante o desenvolvimento, e que deixam o processamento mais devagar. Já a segunda conta com a existência de outros sensores mais estabelecidos no ambiente do Micropython, que não resultem em problemas de leitura ou performance.

Figura 7: BME680
Fonte: https://produto.mercadolivre.com.br/MLB-1591094884-sensor-de-gas-temperatura-presso-e-umidade-bme680-bme-680-_JM

5.4.2 Armazenamento

Dada a simplicidade do projeto, o banco de dados foi pensado como um arquivo database.csv, no qual cada linha corresponde a uma iteração do já citado loop de leitura. Estas se iniciam com o timestamp da medida como índice, e lista cada leitura separada por vírgula, à exceção da primeira linha que mostra os nomes de cada medida. A Figura 8 mostra o começo de uma base de dados de exemplo.

database.csv

```
timestamps,proximity,external_sound,internal_sound,external_temperature,internal_temperature,external_humidity,internal_
1702155689000,0,40,40,26.55285,27.36125,0.0,0.0,1006.14,1010.503,12779099,11995542
1702155970000,0,40,40,26.54601,27.06906,0.0,0.0,1006.143,1010.523,85409,101495
1702155972000,0,40,40,27.9173,27.95519,0.0,0.0,1019.823,1020.556,137245,152841
1702155975000,0,40,40,23.9675,24.04797,0.0,0.0,1020.819,1020.87,159059,161503
1702155977000,0,40,40,24.06828,23.9675,0.0,0.0,1002.102,1001.909,198027,201589
1702155980000,0,40,40,24.45148,24.33117,0.0,0.0,1002.495,1002.547,364248,369542
1702155982000,0,40,40,29.13117,29.12336,0.0,0.0,1003.172,1003.24,418585,456095
1702155985000,0,40,40,29.355,29.28469,0.0,0.0,1010.4,1010.739,490744,487893
1702155987000,0,40,40,29.48644,29.1259,0.0,0.0,1010.438,1010.337,481943,492182
1702155990000,0,40,40,29.61164,29.55891,0.0,0.0,1011.57,1010.385,480220,484725
1702155992000,0,40,40,29.69406,29.63508,0.0,0.0,1011.552,1011.631,499501,497651
1702155995000,0,40,40,29.13586,29.72824,0.0,0.0,1010.813,1011.836,676564,735042
1702155997000,0,40,40,29.16359,29.14367,0.0,0.0,1010.579,1010.669,699093,765180
1702156000000,0,40,40,29.14348,29.16613,0.0,0.0,1009.044,1010.327,773996,805549
1702156002000,0,40,40,29.28469,29.12336,0.0,0.0,1009.045,1009.021,790386,793186
1702156005000,0,40,40,29.34992,29.28508,0.0,0.0,1009.195,1009.13,748972,769563
1702156007000,0,40,40,29.15656,29.34992,0.0,0.0,1008.826,1008.807,791317,783930
1702156010000,0,40,40,29.20539,29.13996,0.0,0.0,1008.844,1008.838,798845,798845
1702156012000,0,40,40,29.47082,29.17375,0.0,0.0,1008.865,1008.835,806517,588806
1702156015000,0,40,40,29.44621,29.45051,0.0,0.0,1009.258,1009.251,805549,595065
1702156017000,0,40,40,29.49367,29.44621,0.0,0.0,1009.57,1009.515,598777,615791
1702156020000,0,40,40,29.52609,29.45051,0.0,0.0,1009.704,1009.717,606343,615791
```

Figura 8: Print de arquivo database.csv de exemplo

A ideia inicial para o deste arquivo envolvia um banco de dados na nuvem, provavelmente utilizando Firebase. Porém, ainda que esta solução provavelmente funcionasse no escopo atual do projeto, a escalabilidade poderia ser comprometida. Isso porque não são conhecidas as redes locais nos quais o projeto pode ser inserido com várias placas, e cada salvamento seria uma requisição ao servidor. Isso poderia sobrecarregar a rede local, ainda que o servidor pudesse aguentar dezenas de requisições.

Assim, decidiu-se pela utilização do leitor de cartão Micro SD já instalado. Por motivos ainda não compreendidos, apenas cartões de 8GB funcionaram, tendo os outros tamanhos

resultado em timeout na biblioteca microsd ou em um erro *"no SD card"*. Porém, mesmo com a maior taxa de salvamento de uma linha por minuto, um único cartão destes consegue armazenar cerca de 180 anos de dados.

Dessa forma, a classe DatabaseManager inicia armazenando os caminhos para os arquivos, tanto do banco de dados como outros que serão discutidos a seguir. Então, utiliza a biblioteca sdcard e o protocolo spi para inicializar a interface com o cartão Micro SD. Por fim, cria o banco de dados se for a primeira inicialização, e salva seu file object como um atributo. Isto é feito para economizar processamento, já que o arquivo do banco de dados é periodicamente acessado, ao contrário dos outros arquivos salvos.

Sobre estes, dois outros arquivos são armazenados no cartão de memória: wifi_credentials.txt e device.txt. Este salva os dados que descrevem a placa, que são: id; name: nome que aparece para a placa no aplicativo; location: local em que a placa se situa; frequencyOfSaving: o timer que foi citado na parte de leitura dos sensores.

Já o arquivo wifi_credentials.txt, como o próprio nome diz, armazena o nome e a senha da rede ao qual a placa deve se conectar. Isso foi feito para que não seja necessário passar novamente pelo procedimento do Bluetooth, relatado na seção a seguir, quando a placa é reiniciada.

Por fim, cabe ressaltar que além de escrever e apagar os arquivos, a classe DatabaseManager também os lê e retorna a string contida. Para o wifi_credentials.txt e o device.txt isso é feito de forma direta, o arquivo é lido como um todo e retornado integralmente. Porém, a base de dados é um caso especial por dois motivos: a necessidade de filtros de timestamp e de leitura em pacotes. O primeiro fator força a leitura do banco linha por linha, convertendo o timestamp de string para inteiro e verificando se está no intervalo entre o filtro inicial e final. Caso seja menor que o filtro inicial, é ignorado; caso esteja no intervalo, a linha é agregada a uma variável; e caso seja maior que o filtro final, ou caso o número máximo de linhas por pacote seja alcançado, o loop é interrompido e a variável contendo todas as linhas aceitas é retornada com um yield.

A parte do número máximo de linhas ocorre por dois motivos: limitação da memória da Raspberry Pi e do número de caracteres que podem ser enviados numa mensagem HTTP 1.1. Em ambos os casos, é necessário carregar os dados aos poucos, o que é feito criando um generator que devolve 80 linhas por vez. O número 80 foi obtido em testes empíricos que resultaram em cerca de 120 linhas por mensagem passando por inteiras. Uma margem de segurança foi adicionada e chegou-se ao valor 80.

5.4.3 Resposta a requisições

Como dito na seção Comunicação, esta é feita majoritariamente por Wi-Fi. Para se conectar a esta, dois caminhos são possíveis: a conexão já foi feita uma vez, as credenciais estão na memória e podem ser utilizadas; ou esta é a primeira conexão nesta rede e o envio de credenciais deve ser feito por Bluetooth.

Para o segundo caso, é utilizado o Bluetooth Low Energy (BLE), que como diz o nome, gasta menos energia elétrica do que o Bluetooth comum[52], algo necessário para uma placa que pode ser alimentada por bateria. Ao ser instanciada, a placa procura pelo já citado arquivo *wifi_credentials.txt*. Caso não exista, ativa o Bluetooth e espera a chegada de

mensagens contendo o nome e a senha da rede Wi-Fi ao qual a placa deve se conectar.

Uma única mensagem bluetooth não consegue carregar o nome e a senha inteiros, o que motivou a utilização de uma sequência de mensagens. Para reconhecer a mensagem como um todo, um padrão foi criado, segundo o qual a mensagem deve conter três barras verticais: uma no início da mensagem, outra no fim, e uma separando o nome da rede da senha. Assim, cada mensagem que chega é adicionada a uma variável local e uma expressão regular tenta reconhecer o padrão estipulado. Em caso negativo, essa variável é reiniciada e o loop continua.

Já em caso positivo, uma tentativa de conexão na rede é feita. A conexão tem 60 segundos para ocorrer, e se isso não ocorrer, supõe-se que a senha inserida estava incorreta. A mensagem "not_connected" é devolvida pelo BLE para o aplicativo, e aguarda-se outras mensagens com novas credenciais. Porém, se a conexão é estabelecida, o IP obtido é enviado ao aplicativo via BLE, as credenciais recebidas são enviadas ao database_manager para serem salvas e um socket na porta 80 é aberto. O socket aberto é passado ao ServerManager, que cuida da conexão a partir deste ponto.

No ServerManger, um loop de espera de requisições é iniciado. O protocolo de comunicação adotado foi o HTTP 1.1, já que uma única conexão por vez é necessária. Requisições GET são enviadas do celular para o microcontrolador e este as trata e devolve a mensagem correspondente, podendo ou não conter dados.

Ao receber uma requisição, o cabeçalho é recortado para obter-se a instrução, que é direcionada para a execução correta pelo método `__do_instruction`. Terminado o processamento, o método `__send_response` é chamado para enviar a resposta ao aplicativo, também no padrão HTTP 1.1. As requisições aceitas são:

- */register*, que recebe os dados a serem armazenados no arquivo *device.txt*
- */device*, que retorna os dados sobre a placa salvos em device.txt
- */data?start=x?end=y*, que devolve as leituras feitas entre os timestamps x e y, como relatado na seção anterior
- */clear*, que apaga as leituras do banco de dados
- */disconnect*, que apaga as credenciais de Wi-Fi e os dados sobre a placa, formatando-a para mudar de localidade. O banco de dados não é apagado

6 Aplicativo

Como mencionado anteriormente, este projeto conta com um aplicativo móvel que estabelece a conexão do microcontrolador com a internet e interpreta os dados provenientes de cada colmeia.

O projeto desenvolvido em 2022[44] contava com um aplicativo Android, porém este estava voltado estritamente a conexões via Bluetooth, o que vai contra o principal objetivo principal do projeto - comunicação via Wi-Fi. Além disso, existia a possibilidade de se comunicar apenas com uma única placa, limitando o poder de controle do usuário.

Tendo em vista que a única funcionalidade que poderia ser aproveitada do código anterior era a visualização de gráficos, foi decidido seguir com a implementação de um novo aplicativo, que conta com requisições HTTP e uma estrutura de cadastramento da placa via Bluetooth.

O aplicativo foi desenvolvido utilizando o framework Flutter, conhecido por sua capacidade de gerar versões nativas tanto para iOS quanto para Android. Para este projeto, focou-se exclusivamente em testes realizados em dispositivos Android.

Embora uma plataforma web seja uma opção totalmente viável, especialmente devido à sua eficácia na visualização e tratamento de dados, optou-se inicialmente por um aplicativo devido à sua praticidade na comunicação Bluetooth durante o processo de cadastramento. Essa decisão também foi influenciada pela continuidade em relação ao projeto anterior, que estava exclusivamente centrado na plataforma Android. Vale destacar que a tecnologia escolhida é cross-platform, proporcionando espaço para evolução futura em direção a uma visualização desktop.

6.1 Objetivos do aplicativo

Em resumo, os objetivos principais do aplicativo são:

1. Cadastrar uma placa à internet: O aplicativo tem como propósito inicial ser capaz de conectar uma placa à internet e criar um tipo de identificação à ela.

2. Oferecer ao usuário uma visualização dos dados coletados pelos sensores na colmeia por meio de um gráfico: Outro objetivo fundamental do aplicativo é proporcionar aos usuários uma visualização clara e acessível dos dados coletados pelos sensores nas colmeias, apresentando essas informações por meio de gráficos intuitivos.

6.2 Tecnologia

Como framework de programação, optou-se por adotar o Flutter[11], uma ferramenta open source desenvolvida pela Google com o propósito de criar aplicações nativas para diversas plataformas a partir de um único código. O Flutter utiliza a linguagem de programação Dart[5], uma linguagem orientada a objetos conhecida por sua performance no desenvolvimento de interfaces de usuário, também desenvolvida pela Google.

A arquitetura do Flutter é estruturada em camadas independentes e inacessíveis entre si[12]. Para este relatório, a camada mais relevante é a Framework, que mantém contato direto com o desenvolvedor. Uma de suas subcamadas cruciais é a de Widgets[31], que descreve partes da visualização da UI, como texto, botões ou páginas inteiras. Esta subcamada desempenha um papel fundamental na compreensão desta seção sobre o aplicativo.

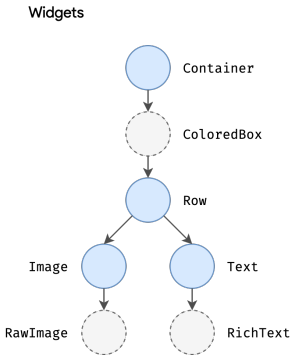


Figura 9: Exemplo de uma árvore de Widgets

A escolha do Flutter leva em conta os seus pontos mais fortes, como o Hot Reload, sua performance, ser cross-platform, comunidade ativa e diversos recursos disponíveis.

O Hot reload é uma funcionalidade do Flutter que permite observar as modificações feitas no código base de forma rápida e fácil a partir de uma UI[22]. Isso é possível graças à utilização da Dart Virtual Machine[6], uma máquina virtual que recebe as atualizações feitas no código e, com a ajuda do framework, atualiza a árvore de Widgets para uma rápida visualização das mudanças.

Apesar do Flutter ter uma performance ligeiramente inferior em comparação com plataformas nativas (Android e iOS), ele supera outras frameworks híbridas, como o React Native[33]. Isso se deve ao uso da linguagem Dart (com uma taxa de compilação superior à do JavaScript do React)[41] e à interação direta do usuário com os componentes, sem a necessidade de pontes de comunicação, ao contrário do React Native[34].

Por fim, ao ter uma comunidade ativa, foi possível ter acesso a bibliotecas capazes de realizar funções de menor baixo nível, como por exemplo, comunicação HTTP, comunicação BLE (Bluetooth Low Energy), salvamento interno e exportação de dados, entre outros.

Apesar da tecnologia escolhida permitir gerar aplicativos cross-platform, seria preciso o acesso a um sistema macOS para conseguir debugar e compilar um aplicativo iOS. Por esse fato, o projeto foca apenas na de Android por ser a única possível de ser testada durante o desenvolvimento. Para iOS não há certeza da garantia de permissões suficientes, como por exemplo uma comunicação por Bluetooth, ou acesso à dados do Wi-Fi do usuário. O teste e compilação de um sistema iOS seria uma possibilidade de trabalho futuro.

6.3 Fluxos

O aplicativo apresenta duas páginas principais acessíveis por meio das abas localizadas na parte inferior: "Dispositivos"e "Dados". Cada uma dessas abas possui fluxos distintos, cujas funcionalidades serão detalhadas nesta seção.

Detalhes mais técnicos do aplicativo serão abordados a partir da seção 6.5, que se concentra principalmente em explicar tecnicamente as funcionalidades desenvolvidas.

6.3.1 Dispositivos

A seção de dispositivos é o ponto central para gerenciar as placas registradas e/ou aquelas encontradas na rede do usuário. Sua página inicial é segmentada em duas áreas distintas: "Meus Dispositivos"e "Dispositivos Online".

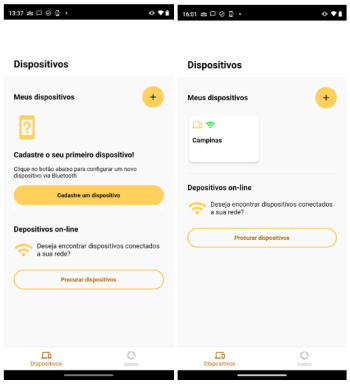


Figura 10: Foto da página de dispositivos. A primeira página é quando o usuário não possui nenhum dispositivo salvo, a segunda é quando ele possui no mínimo um.

No primeiro acesso, em vez de apresentar imediatamente a Figura 10, o aplicativo exibirá uma tela solicitando permissão de localização. Essa etapa inicial é crucial para diversas funcionalidades essenciais do aplicativo, incluindo a inicialização de uma busca por dispositivos Bluetooth, a identificação de redes Wi-Fi disponíveis e, por fim, o cadastro da localização na placa.

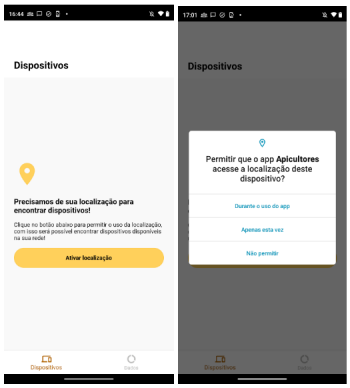


Figura 11: Permissão de localização

O botão "Ativar Localização"abrirá um pop-up no dispositivo móvel do usuário, solicitando permissão para o uso da localização "Durante o uso do aplicativo". Após concedida, todas as funcionalidades dependentes de localização estarão prontamente disponíveis.

Mesmo no caso de o usuário clicar erroneamente em "Não Permitir", há uma solução para contornar a situação:

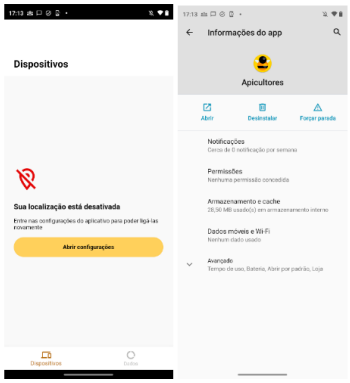


Figura 12: Tratamento de erro: o aplicativo abre a área de permissões do celular

Área "Meus dispositivos"

Essa área possui três objetivos principais: cadastrar um novo dispositivo, gerenciar dispositivos salvos e editá-los caso estejam on-line.

O primeiro passo que o usuário deve seguir é cadastrar um novo dispositivo. Este procedimento é executado assim que a placa é conectada à tomada pela primeira vez. Durante esse processo, a placa é conectada à internet e sua identificação é registrada.

O principal objetivo foi tornar este processo o mais simples possível: toda a configuração e inicialização da placa são realizadas diretamente no celular. Basta instalar a placa na colmeia e conectá-la à tomada. O cadastro inicia ao clicar no botão "Cadastre um Dispositivo"(caso o usuário não tenha dispositivos salvos) ou no ícone de adição ao lado do título "Meus Dispositivos"(Figura 10).

Nesta etapa, é necessário localizar a placa por meio da comunicação Bluetooth, a qual, como explicado anteriormente, estará ativa uma vez que ainda não houve nenhum processo de cadastramento.

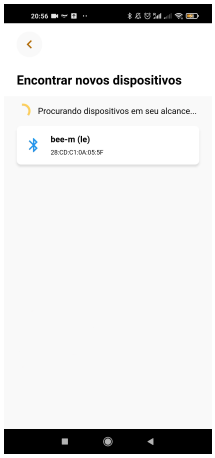


Figura 13: Resultados da pesquisa de dispositivos bluetooth próximos. O da placa sempre terá nome "bee-m"

No decorrer desse processo, serão encontrados todos os dispositivos Bluetooth próximos ao usuário. O dispositivo correto será aquele chamado "bee-m". Ao clicar em seu cartão, uma conexão tentará ser estabelecida. Caso algum erro aconteça nessa etapa, será mostrado um ícone vermelho ao lado do cartão, permitindo que o usuário ainda possa tentar se conectar de novo. No entanto, caso tenha sucesso, o usuário será direcionado ao fluxo de "Conectar a Placa à Internet".

Figura 14: Fluxo onde é pedida a senha do Wi-Fi para conectar a placa à internet

Com o intuito de simplificar o processo para o usuário, evitando a digitação dos dados da rede Wi-Fi, foi utilizada a biblioteca Flutter *network_info_plus*[15] para obter o nome da rede. Contudo, devido à impossibilidade de recuperar a senha, é solicitado que o usuário a insira manualmente. Importante observar que a rede não pode ser de 5GHz, pois tentativas durante o desenvolvimento mostraram que a placa não é compatível com esse tipo de rede.

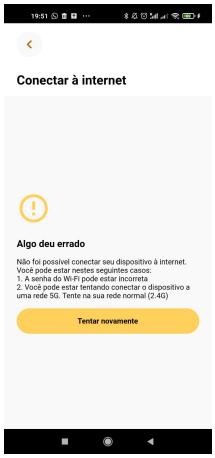


Figura 15: Fluxo onde a placa não consegue se conectar à internet devido algum erro

Após o envio da senha e a bem-sucedida conexão à internet, ocorre a primeira comuni-

cação via Wi-Fi com a placa. Durante essa comunicação, são solicitados os dados da placa. Se houver uma resposta contendo os dados, o usuário conclui o processo de cadastramento e pode iniciar o monitoramento. No entanto, se for uma placa nova, o usuário deve fornecer informações essenciais para a coleta de dados.

Figura 16: Fluxo em que se cadastra os dados de uma nova placa

As informações solicitadas incluem:

- Nome da Placa: Importante para diferenciar as placas monitoradas em diferentes colmeias.
- Localização da Placa: Armazena a latitude e longitude da localização atual do usuário.
- Frequência da Coleta: Permite ao usuário definir a frequência de coleta de dados na placa, indicando um valor inteiro e escolhendo entre segundos, minutos, horas ou dias.

Idealmente, a localização da placa seria determinada automaticamente, mas devido à ausência de GPS na placa, essa responsabilidade foi transferida para o usuário. Neste projeto, é possível apenas cadastrar a localização atual do celular, embora a possibilidade de fornecer uma região através da API do Google Places[26] seja uma ideia interessante para futuras implementações.

O aplicativo exige que o usuário preencha todos os campos e exibirá uma mensagem de erro se algum deles for omitido. Ao preencher corretamente todos os campos, o aplicativo enviará o JSON contendo os dados por meio de uma solicitação POST no endpoint "/register". Se o cadastro for bem-sucedido, o aplicativo apresentará uma tela de sucesso, incentivando o usuário a iniciar o monitoramento das colmeias.

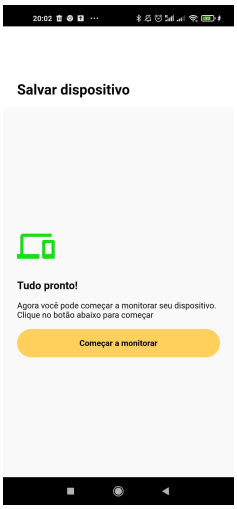


Figura 17: Fim do cadastro de uma placa com sucesso

A última funcionalidade possível nesta seção de "Meus dispositivos"é a edição de dados de cadastramento da placa. Ao clicar no cartão de algum dispositivo salvo presente na visualização 10, a página de "Detalhes do dispositivo"é apresentada. Esta área permite a realização de ações específicas em um dispositivo particular, como alterar seu nome, localização e frequência de coleta. Ao modificar qualquer uma dessas informações, um botão é exibido na tela, permitindo o envio das alterações para a placa.

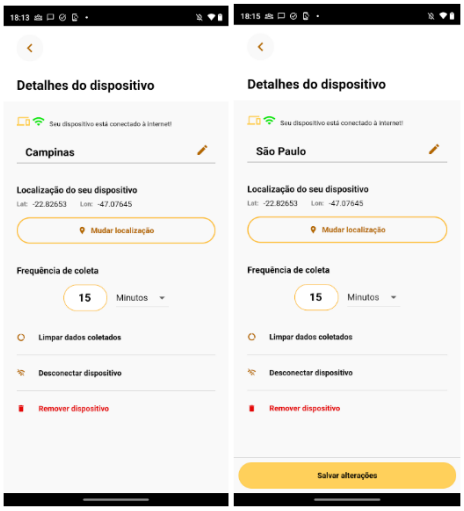


Figura 18: Exemplo dos detalhes do dispositivo Campinas Na segunda foto, como o nome foi editado, o botão de salvar aparece

Além de editar suas informações, é possível realizar três funções adicionais:

- Limpar Dados Coletados: Envia uma requisição ao endpoint "/clear".

- Desconectar o Dispositivo: Envia uma requisição ao endpoint "/disconnect". Devido a limitações da placa, é necessário retirar o dispositivo da tomada e conectá-lo novamente para refazer seu cadastro.

- Remover o Dispositivo: Exclui o dispositivo do aplicativo do usuário, mas não executa nenhuma ação na placa. Ela pode ser encontrada novamente na seção "Dispositivos Online"na página de dispositivos.

Área "Dispositivos on-line"

Outro fluxo possível na área de dispositivos é encontrar uma placa já cadastrada e conectada à rede. Isso pode ser útil caso o usuário tenha apagado o aplicativo ou deseje realizar o monitoramento em outro celular, o que permite que vários celulares sejam capazes de analisar uma mesma colmeia.

Ao clicar no botão "Procurar Dispositivos" na Figura 10, o usuário pode localizar um dispositivo previamente cadastrado. Este processo pode levar alguns minutos, e a explicação para o tempo necessário está detalhada na seção 6.8.2.

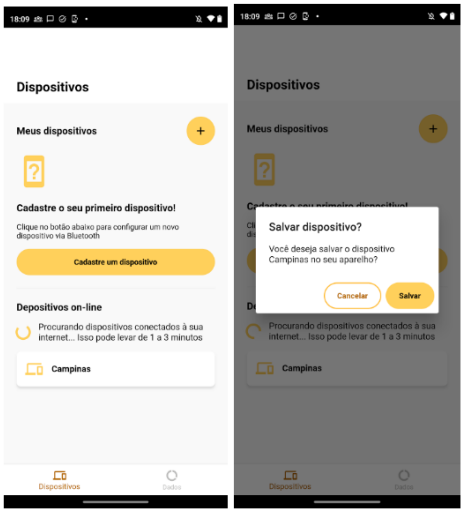


Figura 19: Processo de encontrar uma placa on-line e salvá-la no aplicativo

6.3.2 Dados

Na seção de dados, toda a visualização do que os sensores coletam na colmeia é realizada por meio de gráficos.

Para isso, é necessário escolher qual placa (dentro daquelas conectadas à internet) e qual a faixa de tempo dos dados para a análise. A opção de escolher a faixa de tempo é essencial devido às limitações de memória. Caso todos os dados fossem recuperados a cada vez, o aplicativo poderia falhar, dependendo do tamanho do arquivo recebido. Para possibilitar análises de dados anteriores à data atual, também é permitido escolher datas de início e fim específicas, com o limite de 1 ano entre elas.

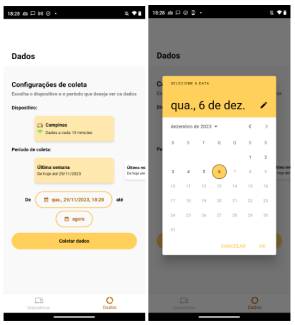


Figura 20: As opções do dispositivo e de datas são apresentadas em carrossel

As opções disponíveis para escolha de faixa de tempo são: última semana, último mês, últimos 6 meses, último ano ou customizada. Ao clicar no botão "Coletar Dados", uma requisição será enviada ao endpoint "/data"contendo timestamps como parâmetros de consulta, referentes à faixa de tempo escolhida.

Gráfico

Para a criação do gráfico, foi amplamente utilizada a biblioteca *syncfusion_flutter_charts*[28]. Esta biblioteca possui uma animação fluida, boa performance e oferece uma ampla variedade de tipos de gráficos. Neste projeto, foi utilizado especificamente o SfCartesianChart.

Junto com a biblioteca, foi desenvolvida uma estrutura de visualização que permite ao usuário controlar a escala de tempo referente aos seus dados. Ele pode observar os dados em relação a uma hora (H), um dia (D), uma semana (S), um mês (M) e um ano (A).

Para controlar a faixa de tempo, o usuário interage com os botões de seta para a esquerda e direita. Por exemplo, ao escolher a escala de tempo (H), ele visualiza os dados desde o início do horário atual (caso fosse 18h30, o inicio é 18h00) até o horário do último dado coletado. Agora, se ele clicar na seta esquerda, verá os dados ainda em uma escala de tempo do tamanho H, mas das 17h até as 18h.

Figura 21: Foto do gráfico ná escala D, do horário 18h até a última medição

Para facilitar ainda mais a leitura do gráfico, é mostrada uma legenda dos dados ao clicar ou arrastar o dedo pela interface. Assim, é detalhado ao usuário qual o valor de tempo selecionado e quais são os valores de cada gráfico sinalizado pela cor.

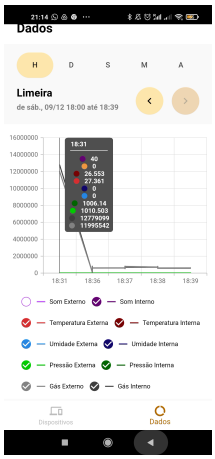


Figura 22: Legenda mostrada acima do gráfico ao tocar sobre sua área

A implementação da estrutura de escala de tempo representou um dos desafios mais significativos no desenvolvimento da visualização em gráficos. Nesse contexto, foi preciso realizar operações de subtração e adição de durações em timestamps, levando em consideração as particularidades de cada escala de tempo. O objetivo central era proporcionar uma experiência fluida e performática, permitindo que o usuário explorasse o gráfico tanto em uma perspectiva macro quanto micro. Essa abordagem foi inspirada no aplicativo Saúde do iOS, como evidenciado na figura abaixo.

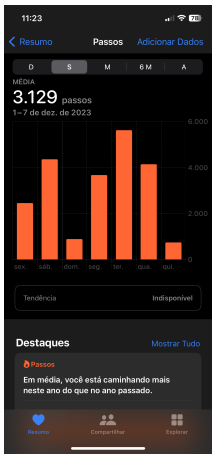


Figura 23: Aplicativo saúde do iOS, com a visualização do gráfico com quantidade de passos.

Como é possível ver na Figura 22, abaixo do gráfico, estão disponíveis checkboxes que facultam ao usuário a escolha dos dados que deseja visualizar. Essa flexibilidade permite a

comparação de diversos tipos de gráficos em uma única visualização.

Além disso, na área abaixo das checkboxes, é apresentado um resumo dos dados atualmente visualizados no gráfico. Esse resumo inclui os valores mínimo e máximo de cada sensor, indicando a data e o horário de cada medição. Uma média dos valores na faixa de tempo observada no gráfico também é exibida.

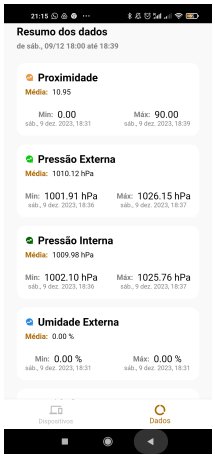
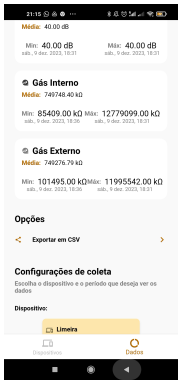
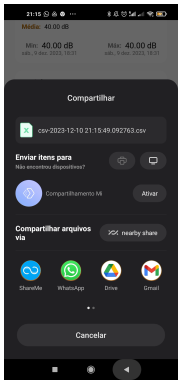


Figura 24: Área de resumo dos dados, com valor de média, mínimo e máximo

Outra funcionalidade implementada é a capacidade de exportar os dados em formato CSV, com o auxílio da biblioteca *csv*[17]. Ao clicar neste botão, todos os dados recuperados da placa são transformados em uma estrutura CSV, prontos para serem compartilhados através de diferentes plataformas, como o Google Drive, por exemplo. Para possibilitar esse compartilhamento, foi incorporada à biblioteca *share_plus*[16]. Essa funcionalidade foi um pedido do professor Roberto Greco, possibilitando o manuseio dos dados também na Web.



(a) Botão "Exportar CSV"



(b) Opções de compartilhamento

Figura 25: Processo de exportação de CSV

Por fim, a opção de configuração de coleta, conforme mostrado na Figura 20, permanece

sempre disponível ao final da página. Isso permite que o usuário solicite faixas de tempo maiores ou dados de outro dispositivo a qualquer momento.

Com essas funcionalidades, todas as metas estipuladas para este aplicativo foram alcançadas. No entanto, é importante destacar que ainda há espaço para evolução, e essas possíveis melhorias serão detalhadas na seção 10 do documento.

6.4 Sistema de Design

Para manter uma consistência na interface, estabelecendo um padrão de comportamento e estilo em todos os fluxos do aplicativo, foi implementado um Sistema de Design[32] neste projeto.

Por exemplo, todos os botões possuem o mesmo estilo, com opções de variantes. Em vez de implementar esse estilo repetidamente, o programador precisa apenas fornecer o texto do botão, a função de callback e a variante desejada. O estilo de fonte, a cor do botão e seu tamanho são propriedades não editáveis, impedindo que o programador quebre o padrão proposto.

Além de manter a consistência, outra vantagem do Sistema de Design é acelerar o processo de desenvolvimento e manter o código mais limpo, pois não é necessário escrever linhas de estilização, apenas as informações necessárias para o funcionamento correto do componente.

A proposta do Sistema de Design é ser um projeto independente, sem qualquer conexão com o código principal. Ele decide o que exportar e esconder para o projeto, impedindo que o desenvolvedor utilize outros Widgets que estejam fora do padrão. A arquitetura do Sistema de Design é inspirada no Atomic Design[53], onde os componentes padrões são separados em átomos, moléculas e organismos.

Para manter esse nível de independência entre o projeto principal e o Sistema de Design, foi utilizada uma pasta de pacotes dentro do repositório, transformando-a em uma biblioteca. Essa pasta não pode importar nada além do que foi instalado em seu *pubspec.yaml* (arquivo de bibliotecas), e, além disso, o pacote seleciona quais arquivos serão exportados.

O pacote exporta toda a biblioteca de Material 3 Design[23] do Flutter. Dessa forma, ao importar a biblioteca do Sistema de Design no código principal, o desenvolvedor não precisa importar no arquivo a biblioteca do Material Design.

```
1      // O único import que precisa para
2      // ter acesso a todos os Widgets do Material Design
3      // e do Design System
4
5      import 'package:design_system/design_system.dart';
```

O projeto principal importa a biblioteca do Sistema de Design em seu pubspec.yaml e deve utilizá-la exclusivamente, em vez do material.dart. Ao importá-la, o projeto tem acesso a todos os Widgets de Material Design, juntamente com os estilos personalizados desenvolvidos exclusivamente para o projeto dos Apicultores.

6.5 Arquitetura

Esta seção explora as decisões arquiteturais relacionadas ao gerenciamento de estados, navegação e armazenamento de textos no aplicativo mobile.

6.5.1 Gerenciamento de Estados (Bloc)

O Flutter oferece diversos padrões de arquitetura amplamente adotados pela comunidade, como Redux, Bloc, Provider, Clean, MVVM e MVC[37]. Para este projeto, optou-se pela arquitetura Bloc.

O Bloc é um padrão arquitetural que visa separar de maneira eficiente a camada de apresentação (UI) da lógica de negócios, resultando em código testável, de fácil compreensão e reutilizável. Esse padrão baseia-se na emissão de um estado a partir de um evento enviado pela UI[30].

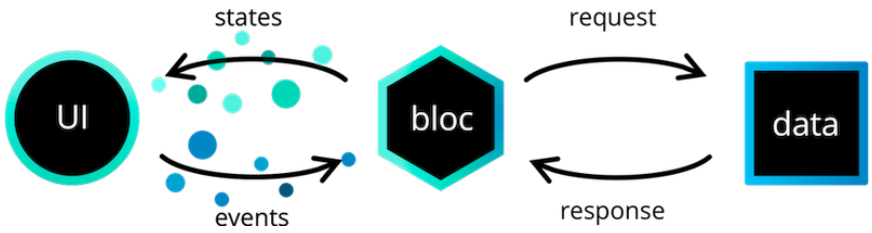


Figura 26: Demonstração do funcionamento do Bloc
Fonte: https://bloclibrary.dev/assets/bloc_architecture_full.png

Essa arquitetura é composta por três componentes principais: Estado (State), Evento (Event) e o próprio Bloc. Dentro do código, esses componentes são representados por classes que se comunicam por meio de instâncias. Para melhorar a clareza e seguir as convenções estabelecidas pela documentação do Bloc[3], o projeto adotou uma nomenclatura consistente.

Para ilustrar esses componentes, considere a situação em que o usuário deseja carregar e visualizar os dados coletados de uma placa específica ao clicar em um botão.

Eventos (Events)
O Event é a classe que sinaliza uma ação requerida da UI para a camada de dados. Por exemplo, ao clicar no botão da página de gráficos, o usuário está indicando ao aplicativo que ele deve se comunicar com a placa e exibir na tela seus dados salvos. A classe criada para este evento é a seguinte:

```
1       class GraphDataFetched extends GraphDataState {
2       final BeeDeviceEntity device;
3       const GraphDataFetched(this.device);
4   }
```

Neste exemplo, a classe contém um atributo "device", que se refere às informações da placa: id, nome e o endereço IP ao qual ela está conectada. Tais informações são necessárias para

realizar a requisição. Portanto, toda informação necessária para realizar qualquer ação na camada de dados é passada pela classe de Event. A nomenclatura sugerida é que os eventos estejam no tempo verbal do passado, uma vez que, na visão do Bloc, são ações que já foram realizadas[3].

Estados (States)
O State é a classe que descreve o estado atual resultante de um evento. Por exemplo, ao receber o evento de carregar os dados, o Bloc primeiro precisa emitir um estado de carregamento. Assim, de acordo com o resultado da requisição, o novo estado emitido será de sucesso ou erro.

```
1  class GraphDataLoading extends GraphDataState {}
2
3  class GraphDataFailure extends GraphDataState{}
4
5  class GraphDataSuccess extends GraphDataState {
6          final GraphDataEntity data;
7          const GraphDataSuccess(this.data);
8  }
```

Neste exemplo, tem-se os três estados mais básicos de um processo de requisição. No estado de Success, há o atributo "data", que é a resposta recebida da camada de dados. Assim, a UI, ao receber a instância deste estado de sucesso contendo os dados, será capaz de montar o gráfico. A nomenclatura sugerida é que os estados sejam apenas substantivos, já que indicam algo que aconteceu pontualmente[3].

Bloc
Por fim, o Bloc é a ponte entre a camada de UI e a de dados. Ele se comunica com a primeira a partir dos eventos e estados, e com a segunda a partir de chamadas assíncronas (async / await).

```
1  class GraphDataBloc extends Bloc<GraphDataEvent, GraphDataState>{
2  final GraphDataUseCase _useCase;
3  GraphDataBloc(this._useCase) : super(GraphDataInitial()) {
4          on<GraphDataFetched>((event, emit) async {
5                  emit(GraphDataLoading());
6                  try {
7                          final data = await _useCase.fetchData(event.device);
8                          emit(GraphDataSuccess(data));
9                  }
10                 catch(_){
11                         emit(GraphDataFailure());
12                 }
13         });
```

```
14      }
15  }
```

No exemplo acima, é mostrado na linha 4 o que é feito ao adicionar o evento `GraphDataFetched` no Bloc. A classe "UseCase"será explicada na seção 6.6.2.

Abaixo é um exemplo de como adicionar um evento em um Bloc

```
1  context.read<GraphDataBloc>().add(GraphDataBlocFetched(device));
```

Na camada de visualização, o código abaixo exemplifica como os estados emitidos do Bloc são recuperados

```
1  return BlocBuilder<GraphDataBloc, GraphDataState>((context, state) {
2          if (state is GraphDataLoading) {
3                  return CircularProgressIndicator();
4          }
5          ...
6  });
7
```

Essa estrutura também é facilmente testável por meio de testes unitários. Infelizmente, devido ao prazo do projeto, foi decidido não escrevê-los, mas o código foi programado com esse objetivo. Com isso, é possível entender o básico de como os estados são gerenciados pelo uso do padrão Bloc, o qual é utilizado em todo o fluxo do projeto.

6.5.2 Navegação

A estrutura de navegação no Flutter, embora bem documentada, às vezes carece de legibilidade e organização no código. Por exemplo, se diferentes partes do código precisarem navegar para a mesma página, geralmente utilizam a seguinte função:

```
1  onPressed: () {
2    Navigator.push(
3      context,
4      MaterialPageRoute(builder: (context) => const SecondRoute()),
5    );
6  }
```

Ao seguir essa abordagem, é fácil que diferentes navegações surjam em várias partes do código, sem possuir um ponto central. Para melhorar esse aspecto, foi adotada a biblioteca Flutter Modular[36]. Nela, é possível navegar por meio de rotas nomeadas, tornando o código mais legível e menos verboso. Um exemplo de navegação é:

```
1  Modular.to.push('/second-route');
```

Todas as navegações possíveis ficam centralizadas no arquivo *app_ module.dart*, e cada uma possui um nome de rota específico, facilitando o controle de telas e permitindo que diferentes partes do código realizem a mesma navegação de forma legível.

6.5.3 Organização de textos

É comum que interfaces de usuário contenham uma variedade de textos informativos. Por exemplo, ao lidar com erros, é essencial fornecer feedback ao usuário sobre a possível razão do ocorrido. No entanto, essa prática pode resultar em códigos Flutter com textos extensos espalhados por toda a camada de visualização, prejudicando a legibilidade e reutilização de textos.

Para evitar esse problema, adotou-se uma abordagem de centralização de Strings com base em um artigo publicado no Medium[40]. Nessa abordagem, são criadas constantes que podem ser acessadas em qualquer parte do código. Veja um exemplo:

```
class Strings {
  Strings._();
  static const String tryAgain = "Tente novamente";
}
```

```
return Button(
  onPressed: () {},
  label: Strings.tryAgain
);
```

Dessa forma, apenas constantes são utilizadas nos Widgets, promovendo um código mais limpo e legível.

6.6 Camadas de arquitetura

A estrutura do aplicativo é organizada em features, como as páginas de dados e dispositivos. Para manter características semelhantes em features distintas, foi implementada uma organização em camadas, divididas em dados (data), lógica de negócios (business logic) e apresentação (presentation). Essa estrutura segue os princípios do padrão Bloc[30], buscando maximizar a separação entre dados e interface do usuário, com a lógica de negócios atuando como a ponte entre eles.

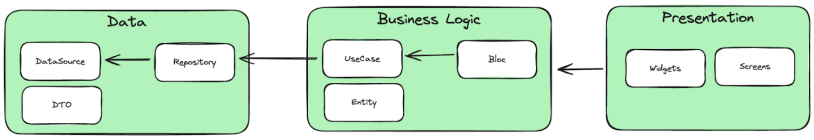


Figura 27: Diagrama sinalizando a comunicação entre as camadas de arquitetura do aplicativo

6.6.1 Camada de dados (Data)

Responsável por realizar requisições HTTP e/ou Bluetooth, interpretando as respostas na linguagem do programa. Compreende os seguintes componentes:

- DTO (Data Transfer Object): Objeto que transporta dados específicos entre um servidor e uma aplicação. Para facilitar a conversão de JSON para esse objeto, foi utilizada a biblioteca *json_ serializable*[21]. A regra é que este objeto contenha apenas métodos de conversão, sendo seu principal objetivo carregar dados.

- Data Source: Classe responsável por conectar-se a um serviço e retornar DTOs em seus métodos. Exemplo: BluetoothDataSource contém métodos para requisições Bluetooth.

- Repository: Camada que contém um repositório de data sources. Por exemplo, para comunicar-se com a placa, são necessários métodos tanto de Bluetooth quanto de HTTP. Os repositórios retornam apenas objetos do tipo Entity, sendo capazes de combinar informações de diferentes DTOs.

6.6.2 Camada de lógica de negócios (Business Logic)

Atua como a ponte entre a camada de dados e a de apresentação, compreendendo os seguintes componentes:

- Entity: Objeto de dados usado a partir desta camada em diante. Pode ser a junção de informações de um ou mais DTOs e pode conter métodos auxiliares não necessariamente ligados ao transporte de dados.

- UseCase: Camada que pode conter um ou mais repositórios. Cada Bloc possui sua UseCase específica. Por exemplo, para procurar dispositivos conectados na rede local, a UseCase precisa do repositório de Wi-Fi para verificar sua ativação no celular do usuário. Com essa confirmação, ela precisará do repositório de dispositivos para realizar uma chamada HTTP caso encontre um conectado à rede. Suas respostas são entregues através de Entity.

- Bloc: Como explicado anteriormente, comunica-se diretamente com a UseCase para realizar requisições à camada de dados. A partir da resposta, emite estados.

6.6.3 Camada de apresentação (Presentation)

Camada que contém apenas Widgets, organizados nas seguintes pastas:

- Screens: são Widgets que representam páginas inteiras. Geralmente contém um Scaffold com uma AppBar.

- Widgets: são componentes mais simples, como um botão, ou uma seção que foi separada do Widget referente a Screen para melhor organização do código.

6.7 Salvamento de dados

No escopo deste projeto, não foi priorizada a implementação de autenticação de usuário nem a utilização de um banco de dados. Dessa forma, todos os dados relacionados ao usuário

que utiliza o aplicativo são armazenados no próprio dispositivo móvel, utilizando o armazenamento interno. Para acessar essa camada no Flutter, optou-se pelo uso da biblioteca *shared_preferences*[27].

Assim, sempre que um usuário cadastra um dispositivo ou o encontra na rede, suas informações em formato JSON são salvas dentro de uma lista de strings armazenada no aplicativo, utilizando o método de chave-valor. Essa biblioteca é direcionada para o armazenamento de dados simples e não críticos. Considerando que as informações de identificação da placa são relativamente pequenas e podem ser recuperadas na própria placa, essa opção foi considerada viável.

Um ponto a ser considerado é que, ao desinstalar o aplicativo, o usuário perde todas as informações guardadas nesse armazenamento, sendo uma limitação dessa solução.

6.8 Comunicação com a Placa

Um dos desafios significativos deste projeto foi estabelecer a comunicação entre os dispositivos envolvidos no sistemas, especialmente ao lidar com recursos de baixo nível, como a interface de comunicação BLE (Bluetooth Low Energy) ou sockets, que não são totalmente implementados por bibliotecas da comunidade. Nesta seção, serão detalhados os desafios, alternativas e decisões tomadas em relação à comunicação com a placa na visão do aplicativo.

6.8.1 Comunicação via Bluetooth

A biblioteca *flutter_blue_plus*[13] foi adotada para realizar a comunicação Bluetooth com o dispositivo. Essa biblioteca opera com Streams[1], onde as informações são enviadas por meio de eventos, e o aplicativo precisa escutá-los para realizar mudanças.

Por exemplo, a cada dispositivo Bluetooth encontrado, a biblioteca envia um evento na Stream, que é escutado pelo Bloc. Dessa forma, os dispositivos aparecem na tela do usuário à medida que são encontrados, em vez de trazer todos os dispositivos de uma vez após um certo período.

Para conectar-se a um dispositivo, uma nova Stream é criada para receber o status da conexão. Essa conexão é monitorada durante todo o processo de cadastro de um novo dispositivo, e se ela se encerrar por algum motivo, o fluxo é interrompido, exigindo que o usuário o reinicie.

Como mencionado anteriormente, é necessário passar os dados do Wi-Fi para a placa por meio do Bluetooth. Esse processo utiliza os conceitos do BLE por meio do perfil GATT[42], que realiza a troca de dados desta conexão por meio de Services e Characteristics. Os passos seguidos são:

1. Encontrar todos os serviços disponíveis do BLE do dispositivo.

2. Selecionar o serviço principal a partir de um UUID definido no código do microcontrolador.

3. Encontrar a Characteristic relacionada à escrita, onde os dados serão enviados.

4. Ao receber todas as Characteristics do serviço selecionado, escolher aquela que possui o UUID especificado no código do microcontrolador como a Write Characteristic.

5. Ao ter acesso a essa Characteristic, a mensagem pode ser passada.

Existem três desafios principais ao enviar esses dados via Bluetooth Low Energy:

1. Padronização da mensagem: Criar um padrão para a mensagem que não conflite com possíveis caracteres da senha do Wi-Fi.

2. Segurança dos dados: Passar esses dados de forma segura.

3. Tamanho da mensagem: Lidar com casos em que o nome do Wi-Fi juntamente com a senha são grandes a ponto de ultrapassar o limite máximo da comunicação BLE (20 bytes).

Para o desafio 1, optamos por separar os dados com o caractere "|", que, por senso comum, é raramente escolhido para criar senhas, tornando-o uma escolha segura. Quanto ao desafio 2, pensamos em criar alguma forma de criptografia que só pudesse ser decodificada entre a comunicação com o aplicativo e a placa. No entanto, como a criptografia não era um objetivo do projeto, esse ponto foi considerado uma funcionalidade futura.

Quanto ao desafio 3, foi necessário dividir os dados do Wi-Fi em pequenos pacotes que não ultrapassassem 20 bytes. O tamanho máximo foi de 17 caracteres por pacote. Esses pacotes são enviados sequencialmente e reunidos no microcontrolador, conforme explicado na seção 5.4.3. É importante ressaltar que as mensagens são transmitidas por meio de uma lista de unidades de código UTF-16, que é a linguagem de comunicação do Bluetooth.

Após passar os dados do Wi-Fi, um novo listener Bluetooth é criado para receber mensagens da placa. Isso é necessário para receber o feedback da conexão da placa ao Wi-Fi. Se a mensagem recebida for *"not_connected"*, significa que não houve sucesso, enquanto o contrário significa que algum endereço IP foi recebido. Em outras palavras, uma Stream é criada sempre que se deseja ouvir dados de uma comunicação BLE. Os passos para a criação dessa Stream são:

1. Seguir os mesmos dois primeiros passos do processo de escrita.

2. Selecionar a Characteristic relacionada à leitura, em que o UUID é especificado pela placa.

3. Instruir essa Characteristic a notificar a Stream quando seu valor for modificado.

4. Retornar a Stream, convertendo o dado recebido de unidades de código UTF-16 para uma String.

Por fim, ao receber com sucesso o endereço IP em que a placa está conectada à internet, é possível fechar a conexão Bluetooth e iniciar toda comunicação através de sockets.

6.8.2 Protocolo HTTP

Após a inicialização da placa, todas as requisições são efetuadas por meio do protocolo HTTP/1.1. Para viabilizar essa comunicação, a biblioteca http[7] foi empregada, uma vez que já implementa os principais métodos de uma API RESTful, com foco especial em GET e POST, que são os métodos utilizados no aplicativo.

A escolha dessa biblioteca decorreu da sua facilidade de uso e da capacidade de ocultar detalhes de baixo nível, como a abertura de sockets, simplificando, assim, a implementação da comunicação por HTTP.

A seguir, é apresentado um exemplo de requisição GET para o microcontrolador, ilustrando a simplicidade proporcionada por essa biblioteca:

```
Future<BeeDeviceDTO> getDeviceData(String deviceIp) async {
    print("trying to connect to device");
    final response = await http.get(
      Uri.parse('http://$deviceIp/device'),
      headers: <String, String>{
        'Content-Type': 'application/json; charset=UTF-8',
      },
    );
    if (response.statusCode != 200) {
      print('Device not found because of ${response.statusCode}');
      throw const BeeDeviceConnectionGetDataException();
    }
    return BeeDeviceDTO.fromJson(jsonDecode(response.body));
  }
```

Entretanto, a utilização dessa biblioteca não eliminou a necessidade de abrir conexões sockets para a comunicação com a placa. Isso foi requerido nos fluxos de encontrar uma placa conectada à rede e no recebimento de dados coletados pelos sensores.

Encontrar uma placa conectada à rede

Para sistemas distribuídos, o Multicast[43] é um protocolo de comunicação recomendado, permitindo o envio de dados de uma única fonte para um grupo de receptores. A maneira mais eficiente de localizar placas conectadas à rede local seria enviar uma mensagem do aplicativo para um endereço específico. Todas as placas que escutassem esse endereço multicast poderiam responder à mensagem, enviando seus dados de volta. No entanto, devido à recenticidade desse protocolo e ao fato de o Flutter ser uma tecnologia relativamente nova, não foram encontradas soluções práticas para realizar essa comunicação.

Diante dessa situação, e considerando que essa funcionalidade é de escopo reduzido em comparação às outras projetadas para o aplicativo, optou-se pela solução mais simples: conectar-se a cada uma das 255 subnets da rede local. Assim, um socket HTTP/1.1 é aberto com um tempo de timeout de 200 ms, percorrendo as subnets de 1 a 255. Caso a conexão seja estabelecida, o socket é encerrado, seguido por uma chamada GET para recuperar as informações do dispositivo.

Nesse fluxo, Streams[1] foram empregadas para a emissão de eventos. Cada vez que uma requisição é bem-sucedida, um evento contendo informações da placa é emitido e apresentado na tela do usuário. Essa abordagem evita que o usuário precise esperar até que todas as subnets sejam requisitadas para visualizar o que foi encontrado durante o processo. Essa solução foi inspirada na biblioteca deprecada network_discovery[24] e está incorporada a este projeto (ver referências[10]).

Contudo, ao multiplicar 200 ms pelas 255 chamadas realizadas, totaliza-se quase 1 minuto de espera para essa pesquisa. Tal experiência do usuário é insatisfatória. Para mitigar essa situação, além das informações da placa, o IP ao qual o dispositivo se conectou ao ser cadastrado é armazenado no armazenamento interno do aplicativo. Dessa forma, caso a placa permaneça conectada à mesma rede, sem alterações que a levem a mudar de endereço IP, o usuário consegue receber o feedback de sua conexão ao abrir o aplicativo, tornando a funcionalidade de pesquisa uma segunda opção.

Receber dados dos sensores

Como anteriormente explanado, não foi viável enviar todos os dados salvos dos sensores no microSD em uma única requisição devido às limitações de memória da placa. Portanto, uma comunicação HTTP simples de pergunta e resposta por meio da biblioteca não foi suficiente para receber dados e plotar o gráfico. Nesse sentido, um socket TCP foi aberto, ainda na porta 80, para receber mensagens e concatená-las em uma String até o encerramento da conexão com o servidor. Esse auxiliar foi denominado DataChunksCollector e pode ser encontrado nas referências[8]. Considerando que o que está sendo recebido é um CSV, foi necessário convertê-lo em um DTO para manter o padrão proposto pelo projeto. Um dos efeitos colaterais dessa solução foi um pequeno aumento no delay do recebimento de dados. Além disso, há o risco de a conexão cair no meio do envio de dados, o que poderia corrompê-los ao chegar no aplicativo.

Com isso, foi detalhado na visão do aplicativo todas as formas de comunicação implementadas. É esperado que no futuro tais comunicações sejam simplificadas ao conseguir trabalhar com um servidor e um banco de dados.

6.9 Bibliotecas externas

Durante toda redação da seção de aplicativo, foram citadas algumas bibliotecas utilizadas para realizar determinadas funções. Esta seção resume todas elas juntamente com as referências.

- flutter_modular[36]: utilizada para criar rotas nomeadas dentro do aplicativo.
- flutter_bloc[30]: utilizada para criar as estruturas arquiteturais Bloc.
- equatable[9]: utilizada para sobrescrever a função "equalsTo"de classes para que objetos possam ser comparados.
- flutter_localizations e intl[20]: utilizada para traduzir os timestamps para português do Brasil.

- flutter_blue_plus[13]: responsável por implementar as funções BLE
- permission_handler[25]: responsável por pedir permissões de localização para o usuário
- connectivity_plus[14]: escuta o status de conectividade do Wi-Fi.
- network_info_plus[15]: responsável por recuperar dados do Wi-Fi que o usuário está atualmente conectado.
- geolocator[18]: responsável por conseguir a latitude e longitude do dispositivo móvel.
- uuid[29]: gera um uuid único para a placa
- http[7]: utilizada para realizar comunicações HTTP com GET e POST.
- json_annotation e json_serializable[21]: para gerar funções que convertem JSON em classes.
- shared_preferences[27]: para guardar dados no armazenamento interno do dispositivo móvel.
- syncfusion_flutter_charts[28]: para plotar os gráficos.
- carousel_slider[4]: utilizado para criar o efeito de carrossel juntamente com animações.
- share_plus e path_provider[16]: utilizado para compartilhar os dados csv via Google drive.
- csv[17]: utilizado para criar um arquivo csv a partir de uma lista de strings.
- flutter_launcher_icons[19]: um auxiliador para criar ícones para o aplicativo.

7 Documentação

A seguir, são listados os repositórios e outras fontes referentes a aos componentes do projeto.

- O repositório contendo o software em Micropython do microcontrolador é encontrado na referência[49]. O repositório é publico e o código final e atualizado se encontra na branch 'main'.
- O repositório contendo o aplicativo em Flutter é encontrado na referência[47]. O repositório é publico e o código final e atualizado se encontra na branch 'main'.
- Vídeos, fotos e outras informações do projeto podem ser encontradas no drive público da referência[50].
- O documento referente a placa contendo suas especificidades é encontrado na referência[38]

8 Resultados

Como resultado final obteve-se uma placa capaz de realizar comunicação via Bluetooth e Wi-Fi, bem como realizar medições em um intervalo de tempo determinado, porém com sucesso em apenas um dos tipos de sensores - o de proximidade. Além disso, foi programado um aplicativo mobile capaz de conectar a placa à internet, realizar sua identificação e comunicar-se via HTTP para receber e apresentar graficamente dados guardados pelo microcontrolador.

Foram gravados três vídeos referentes ao resultado do projeto, e eles podem ser encontrados na referência[51].

- Vídeo *Cadastro-nova-placa.mp4*: É realizado o processo de cadastro com a placa. Nele é possível ver que encontrar dispositivos bluetooth, conectar a placa à internet e editar seus campos acontece de forma rápida e bem sucedida.

- Vídeo *Visualização-gráficos.mp4*: É realizada a chamada à placa para visualizar o gráfico. Visto que a placa consegue realizar comunicações apenas no cenário em que o BME680 não é ligado, não seria possível trazer dados reais de sensores diversos para demonstrar as funcionalidades do aplicativo. Para isso, foi feito um script na placa que gerasse dados forçados de todos os sensores em prol apenas da demonstração do aplicativo. É importante ressaltar novamente que resultados reais dos dados são possíveis apenas com o sensor de proximidade. O aplicativo contou com diversas visualizações importantes dos dados, um gráfico performático e editável, bem como uma funcionalidade útil de exportação de dados.

- Vídeo *Encontrar-dispositivos-on-line.mp4*: É utilizada a funcionalidade de encontrar placas conectadas a rede. No vídeo é possível ver que não há sucesso na primeira vez, porém na segunda tentativa foi possível recuperar a placa. Provavelmente o servidor da placa não respondeu em até 200ms a tentativa de conexão do aplicativo, resultando em um time-out. Isso indica que talvez este valor nem sempre será satisfatório em algumas redes.

O aplicativo pode ser instalado na referência[48] através do seu arquivo ".apk". A placa pode ser obtida através do Prof. Roberto Greco no Instituto de Geociências.

9 Modo de uso

Ao receber em mãos a placa, com os sensores e um cartão microSD, é necessário realizar as conexões, caso elas já não estejam feitas. Uma foto indicando as conexões feitas se encontra no drive de fotos do projeto[50]. É importante também que o microSD seja de 8GB, esteja formatado em FAT32 e com nenhum conteúdo salvo.

Com isso, é possível instalar o sensor em uma colmeia e ligá-lo à tomada. O próximo passo é instalar o aplicativo e cadastrar essa nova placa para que ela comece a realizar suas medições, seguindo todo fluxo mostrado na seção 6.3. Após um tempo, será possível observar as medições da placa dentro do aplicativo.

Em caso de algum erro, é recomendado realizar tentativas esporádicas nas funcionalidades desejadas. Caso o erro persista por mais de 1 dia, será necessário desconectar a placa da internet através da página de "Detalhes do dispositivo" (Figura 18) e realizar seu cadastro novamente via Bluetooth.

10 Conclusão e Trabalhos Futuros

Neste projeto foi possível criar uma sistema IoT, em que uma placa com um microcontrolador Raspberry Pi Pico W conectada a sensores fosse capaz de realizar medições esporádicas do ambiente, bem como trabalhar como um servidor HTTP. Além disso, foi programado um aplicativo Android capaz de gerenciar diversas placas em um só aparelho, imprimir os dados requisitados de acordo com as preferências do usuário e também exportá-los em outras plataformas. A comunicação do aplicativo com a placa via Wi-Fi, apesar de possíveis erros de conexão, ocorre em sua maioria de forma satisfatória, atendendo o principal objetivo do projeto.

Todavia, não foi possível trabalhar com todos os sensores fornecidos. Devido às dificuldades explicadas anteriormente, somente o sensor de proximidade funcionou de forma satisfatória. Assim, o principal trabalho futuro para a placa é fazê-la funcionar juntamente com os sensores de som e o BME680, com o foco em estudar a fundo as bibliotecas existentes e entendendo o motivo de não terem compatibilidade com o projeto.

Além disso, outro trabalho para a placa é conseguir melhorar a memória do seu microcontrolador. Durante o desenvolvimento do software da placa, foram enfrentados diversos problemas relacionados à baixa capacidade de memória do Raspberry Pi Pico W, levando a tomada de decisões que contornam o problema, mas não os resolvem. Como por exemplo, enviar os dados em pequenos pacotes, ou não conseguir trabalhar com o BME680.

Para o aplicativo, cabe o estudo de evoluí-lo para uma visualização desktop na Web ou em um executável, ainda mantendo o código fonte em Flutter. Esta opção facilitaria a visualização dos dados da placa. Além disso, é interessante projetar uma comunicação Multicast para o encontro de placas conectadas à rede, evoluindo o que está programado atualmente e mostrando as placas on-line de forma mais rápida.

Um outro ponto importante de evolução é salvar dados climáticos fornecidos pela API OpenWheaterMap[45] juntamente com dados do sensores, para que seja possível realizar estudos mais valiosos em relação ao comportamento das abelhas. Atualmente os valores de latitude e longitude do aparelho que cadastra a placa são salvos dentro do microSD, sendo possível obter informações climáticas de uma região específica. Este foi um pedido do Prof. Roberto Greco, que infelizmente, não entrou como um dos objetivos do projeto devido a limitação de tempo de seu desenvolvimento.

Por fim, é importante a implementação de testes unitários no aplicativo a fim de manter o código seguro de erros indesejados, e aumentar a qualidade do projeto. Como a criação de diversas funcionalidades em um novo app foi complexa e trabalhosa, é importante que em próximas evoluções haja espaço para a escrita de testes. A arquitetura utilizada, como explicada anteriormente, tem como um dos objetivos ser facilmente testada, o que motiva a implementação destes testes.

Referências

[1] *Asynchronous programming: Streams*. Disponível em: `https://dart.dev/tutorials/language/streams`.

[2] *Blesimpleperipheral*. Disponível em: `https://github.com/micropython/micropython/blob/master/examples/bluetooth/ble_simple_peripheral.py`.

[3] *Bloc: Naming conventions*. Disponível em: `https://bloclibrary.dev/#/blocnamingconventions`.

[4] *carousel_slider*. Disponível em: `https://pub.dev/packages/carousel_slider`.

[5] *Dart*. Disponível em: `https://dart.dev/`.

[6] *Dart: The platforms*. Disponível em: `https://dart.dev/overview#platform`.

[7] *dart.dev: http*. Disponível em: `https://pub.dev/packages/http`.

[8] *Data chunks collector*. Disponível em: `https://github.com/marcoscunharosa/apicultores_app/blob/main/lib/shared/adapter/data_chunks_collector.dart`.

[9] *Equatable: Simplify equality comparisons*. Disponível em: `https://pub.dev/packages/equatable`.

[10] *Exemplo do network discovery implementado*. Disponível em: `https://github.com/marcoscunharosa/apicultores_app/blob/main/lib/shared/adapter/network_discover.dart`.

[11] *Flutter*. Disponível em: `https://flutter.dev/`.

[12] *Flutter architectural overview*. Disponível em: `https://docs.flutter.dev/resources/architectural-overview`.

[13] *Flutter blue plus*. Disponível em: `https://pub.dev/packages/flutter_blue_plus`.

[14] *Flutter community: connectivity_plus*. Disponível em: `https://pub.dev/packages/connectivity_plus`.

[15] *Flutter community: network_info_plus*. Disponível em: `https://pub.dev/packages/network_info_plus`.

[16] *Flutter community: share_plus*. Disponível em: `https://pub.dev/packages/share_plus`.

[17] *Flutter: csv*. Disponível em: `https://pub.dev/packages/csv`.

[18] *Flutter geolocator plugin*. Disponível em: `https://pub.dev/packages/geolocator`.

[19] *Flutter launcher icons*. Disponível em: `https://pub.dev/packages/flutter_launcher_icons`.

[20] *flutter_localizations library*. Disponível em: https://api.flutter.dev/flutter/flutter_localizations/flutter_localizations-library.html.

[21] *google.dev: json_serializable*. Disponível em: https://pub.dev/packages/json_serializable.

[22] *Hot reload*. Disponível em: https://docs.flutter.dev/tools/hot-reload.

[23] *Material components widgets*. Disponível em: https://docs.flutter.dev/ui/widgets/material?gclid=CjwKCAiA98WrBhAYEiwA2WvhOo10obBr229lUl8547H5ep1NrRk9Rbk6xmDfD5gTtuBwE&gclsrc=aw.ds.

[24] *network_discovery*. Disponível em: https://pub.dev/packages/network_discovery.

[25] *permission_handler*. Disponível em: https://pub.dev/packages/permission_handler.

[26] *Plataforma google maps: Places api*. Disponível em: https://developers.google.com/maps/documentation/places/web-service/overview?hl=pt-br.

[27] *Shared preferences plugin*. Disponível em: https://pub.dev/packages/shared_preferences.

[28] *Syncfusion: Flutter charts*. Disponível em: https://www.syncfusion.com/flutter-widgets/flutter-charts.

[29] *uuid*. Disponível em: https://pub.dev/packages/uuid.

[30] *Why bloc?* Disponível em: https://bloclibrary.dev/#/whybloc.

[31] *Widget class*. Disponível em: https://api.flutter.dev/flutter/widgets/Widget-class.html.

[32] M. Bakir, *The benefits of a design system: Making better products, faster*. Disponível em: https://www.toptal.com/designers/design-systems/benefits-of-design-system.

[33] I. Demedyuk and N. Tsybulskyi, *Flutter vs react native vs native: Deep performance comparison*. Disponível em: https://inveritasoft.com/blog/flutter-vs-react-native-vs-native-deep-performance-comparison.

[34] C. Deshpande, *Flutter vs. react native: Which one to choose in 2023?* Disponível em: https://www.simplilearn.com/tutorials/reactjs-tutorial/flutter-vs-react-native#:~:text=ISTRegister%20Now-,Performance,with%20the%20device%27s%20native%20components, 18/08/2023.

[35] R. T. Fielding, M. Nottingham, and J. Reschke, *HTTP Semantics*. RFC 9110, June 2022.

[36] Flutterando, *Flutter modular*. Disponível em: https://pub.dev/packages/flutter_modular.

[37] FlutterTech, *Discover powerful architecture patterns in flutter for scalable and testable apps*. Disponível em: https://medium.com/@FlutterTech/discover-powerful-architecture-patterns-in-flutter-for-scalable-and-testable-apps-5cf83d

[38] F. Fruett and R. Greco, *Multiple sensor beehive monitoring*. Disponível em: https://docs.google.com/document/d/1Mz-MCR6UlPnqEdyOTakaD4TGH3IIKm6izRvWRS6cQM8/edit, Fevereiro 2023.

[39] R. Hammelrath, *Micropython driver for a bme680 breakout*. Disponível em: https://github.com/robert-hh/BME680-Micropython.

[40] M. Hassan, *Handling flutter strings like a pro*. Disponível em: https://maruf-hassan.medium.com/handling-flutter-strings-like-a-pro-22653ea4fd93.

[41] M. Joshi, *Flutter vs react native: A comparison*. Disponível em: https://www.browserstack.com/guide/flutter-vs-react-native, 05/05/2023.

[42] MartinBL, *Bluetooth low energy services, a beginner's tutorial*. Disponível em: https://devzone.nordicsemi.com/guides/short-range-guides/b/bluetooth-low-energy/posts/ble-services-a-beginners-tutorial.

[43] metaswitch, *What is multicast ip routing?* Disponível em: https://www.metaswitch.com/knowledge-center/reference/what-is-multicast-ip-routing.

[44] G. J. S. Moraes, V. A. M. Dantas, A. C. L. C. C. F. Renoldi, H. F. Zimerman, P. P. Alves, J. V. F. Costa, L. S. L. Carmo, and L. F. Bittencourt, *Monitoramento de colmeias com internet das coisas*, (2022).

[45] OpenWeather, *Weather api*. Disponível em: https://openweathermap.org/api.

[46] peterhinch, *Sdcard*. Disponível em: https://github.com/peterhinch/micropython-vs1053/blob/master/sdcard.py.

[47] M. C. Rosa, *apicultores_app*. Disponível em: https://github.com/marcoscunharosa/apicultores_app.

[48] ——, *Instalável (apk) do aplicativo desenvolvido neste projeto*. Disponível em: https://drive.google.com/file/d/1aJJdnQAQ0UamA5S7bSPz-5KcuAxErU1P/view?usp=sharing.

[49] M. C. Rosa and L. C. Castello, *bee_monitoring_micropython*. Disponível em: https://github.com/marcoscunharosa/bee_monitoring_micropython.

[50] ——, *Drive com fotos e vídeos do projeto*. Disponível em: https://drive.google.com/drive/folders/1cbUiAPzULf1b4LdsAIevUsNh2JMjBraG?usp=sharing.

[51] ——, *Vídeos demonstrativos do projeto*. Disponível em: `https://drive.google.com/file/d/11zskwhf-gUlRRcdNpX-72FsxKMWzKoLM/view?usp=sharing`.

[52] UFRJ, *Bluetooth low energy*. Disponível em: `https://www.gta.ufrj.br/ensino/eel879/trabalhos_vf_2012_2/bluetooth/ble.htm`.

[53] A. Wirtups, *Principles of atomic design*. Disponível em: `https://medium.com/galaxy-ux-studio/principles-of-atomic-design-7b03a30c3cb6`.