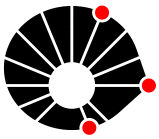

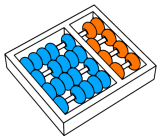

Monitoramento de Colmeias com Internet das Coisas

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 *L. F. Bittencourt*

Relatório Técnico - IC-PFG-22-37
Projeto Final de Graduação
2022 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

Monitoramento de Colmeias com Internet das Coisas

Gabriel J S Moraes* Vinicius A M Dantas† Ana Clara L C C F Renoldi‡
Henrique F Zimerman§ Pedro P Alves¶ Joao V F Costa‖
Lucas S L Carmo** Luiz F Bittencourt††

Resumo

Este trabalho é um relatório do projeto realizado em parceria com uma atividade de extensão do Instituto de Geociências (coordenado pelo Prof. Roberto Greco), cujo objetivo é projetar e implementar um sistema utilizando Internet das Coisas para o monitoramento de colmeias de abelhas nativas. As colmeias serão instaladas no Assentamento Milton Santos (Americana/SP) com o objetivo principal de polinização da vegetação presente.

Após visitar o local e reunir os requisitos do projeto, foi desenvolvido um sistema para coleta de dados da colmeia (temperatura, umidade e som) e um aplicativo de celular para leitura dos dados. Como o assentamento não possui internet, os sistemas se comunicam por Bluetooth.

1 Introdução

Sistemas IoTs são utilizados em diversas esferas da sociedade nos dias atuais. Um sistema IoT é composto por dispositivos de computadores que estão ligados entre si, com a capacidade de transferir dados por uma rede sem a necessidade de uma interação humano-humano ou humano-computador. Além disso, os dispositivos que compõem esse tipo de sistema são dispositivos inteligentes habilitados para a Web, usando sistemas incorporados como sensores, processadores, hardware de comunicação, etc. Isso tudo para enviar e coletar dados, e agir com base em dados adquiridos [1].

Existem alguns desafios para desenvolver e implementar sistemas de Internet das Coisas. Para a constante transferência de dados que esse tipo de sistema proporciona, é imprescindível uma boa conectividade, e ainda existem locais que não possuem conexão à internet.

*g197382@dac.unicamp.br
†v188092@dac.unicamp.br
‡a193858@dac.unicamp.br
§h217771@dac.unicamp.br
¶p204729@dac.unicamp.br
‖j199818@dac.unicamp.br
**l202110@dac.unicamp.br
††bit@ic.unicamp.br

Aqui no Brasil não é diferente, a infraestrutura não alcançou a tecnologia, poucos locais possuem sinal 5G e o acesso a internet no geral é caro.

Para o projeto em questão, foi proposta a implementação de um monitor para colméias, utilizando um sistema IoT para coleta de dados (temperatura, umidade e som) em tempo real. O local em que o projeto será desenvolvido é o assentamento Milton Santos (Figura 1), localizado na cidade de Americana. Assentamentos Rurais são um conjunto de unidades agrícolas, cada uma delas chamada de parcela ou lote, destinadas a famílias de agricultores ou trabalhadores rurais sem condições econômicas de adquirir um imóvel rural [2]. As colméias de abelhas nativas a serem monitoradas serão instaladas no assentamento para o auxílio da comunidade e das famílias residentes, polinizando a vegetação local. O monitor servirá para que os moradores consigam acompanhar a coleta de dados e realizar manutenções necessárias.

Figura 1: Assentamento Milton Santos - Americana/SP
Fonte: Elaborado pelos Autores

Após a visita ao local para extração de requisitos do projeto, foi observado que não havia acesso à internet. Além disso, os futuros usuários do sistema não possuem muita experiência com tecnologia. Dessa forma, para atender as demandas do local, optou-se por um sistema com comunicação bluetooth. A desvantagem de não se usar um sistema IoT é não receber em tempo real os dados, mas para a aplicação em pequena escala do projeto, o bluetooth atende as demandas. Assim, a tarefa é facilitada para os moradores, bastando se conectar ao bluetooth do monitor e extrair os dados (que serão coletados com intervalo de 5 minutos).

Este relatório está dividido nas seguintes seções: Objetivo, que descreve os objetivos planejados para o projeto; Metodologia, que descreve brevemente a progressão da organização e das tomadas de decisão feitas pelo grupo; Implementação Microcontrolador e Implementação Aplicação Mobile, nas quais é descrito detalhadamente a arquitetura utilizada, a estrutura dos códigos e as motivações para cada forma de implementação, tanto para o microcontrolador quanto para o aplicativo; Resultados, descrevendo e apresentando o desfecho do desenvolvimento do projeto; Documentação, que traz instruções de como executar o projeto, desde a montagem do circuito até a coleta dos dados através de um *smartphone*, e, por fim, são apresentadas as conclusões e considerações finais do projeto, juntamente com sugestões de melhorias e as referências bibliográficas.

2 Objetivo

Depois do levantamento de requisitos no local em que as colméias serão instaladas, o objetivo desse trabalho é projetar e implementar um sistema que:

- Coleta dados de temperatura, umidade e som através de sensores.
- Através de um microcontrolador, armazena os dados coletados em um cartão MicroSD.
- Exporta dados armazenados via bluetooth.
- Possui aplicação mobile que conecta via bluetooth com o microcontrolador e recebe os dados armazenados.
- Possibilita a visão das métricas de forma individual ou agrupadas em gráficos.

3 Metodologia

Inicialmente, foi feita a separação do projeto e dos integrantes em duas frentes: Microcontrolador, grupo composto pelos alunos Ana Clara Renoldi, Gabriel Moraes, Henrique Zimerman e Pedro Alves; e Aplicação Mobile, grupo composto pelos alunos João Costa, Lucas Carmo e Vinicius Dantas. Através de reuniões do grupo, foram definidas as especificações gerais de cada parte, assim como um diagrama com o fluxo de dados geral, apresentado na Figura 2.

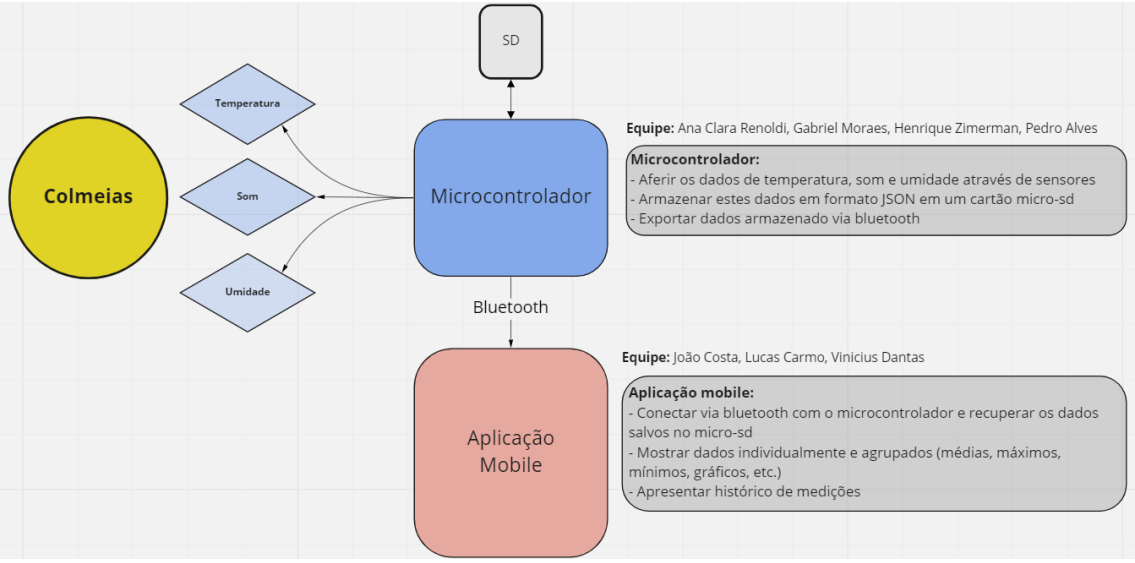

Figura 2: Diagrama fluxo de dados geral
Fonte: Elaborado pelos Autores (Plataforma Miro)

Após a elaboração do diagrama de fluxo de dados geral e separação das equipes, foi dado início à parte prática do projeto. Para o microcontrolador, foram feitas pesquisas de projetos semelhantes[3][4][5] que pudessem ser usados de referência, além de serem listados e cotados todos os equipamentos necessários para o desenvolvimento do circuito. Para a parte da aplicação, se deu início na prototipagem do design inicial do aplicativo e na formulação do código.

4 Implementação Microcontrolador

4.1 Fluxo Geral

A implementação do código do projeto arduino se baseia em duas funções principais: setup e loop. A primeira é responsável pelas configurações iniciais dos módulos do circuito, garantindo que tudo esteja devidamente inicializado e preparado para executar suas funções. Ela é executada uma única vez, no início da execução do programa. Já a função loop é executada sequencialmente *ad infinitum*, sendo ela a responsável por toda a lógica de funcionamento dos módulos.

Neste projeto, a cada 5 minutos são aferidos os valores de temperatura interna, temperatura externa, umidade interna, umidade externa, volume médio de som e *timestamp*. Os valores "internos" e "externos" tem como referência a colméia. Com isso, tais valores são salvos, em formato json, num arquivo de nome "yyyy-mm-dd", em que 'yyyy' é o ano, 'mm' é o mês e 'dd' é o dia em que os valores foram aferidos. Todas as medições de um dia são anexadas no arquivo referente ao dia atual, e esses arquivos são salvos num cartão SD.

Para devolver os dados, o microcontrolador espera receber do aparelho conectado (via bluetooth) uma mensagem contendo apenas o caractere 'g'. Recebida esta mensagem, os dados dos últimos 7 dias, ou seja, os 7 arquivos mais recentes, são enviados para o aparelho.

4.2 Estrutura do Código

4.2.1 Função Setup

Na função setup, inicializa-se os módulos da placa de acordo com o uso que se espera. São setados valores iniciais para os módulos DHT (Digital Temperature and Humidity Sensor), RTC (Real Time Clock) e SD, fazendo verificações de atividade e validez dos mesmos e informando caso haja algum erro. Caso todos estejam funcionando normalmente, dá-se início à execução da função loop.

4.2.2 Função Loop

Na função loop, um contador de tempo é iterado para verificar qual é o tempo decorrido desde a última medição. Ao verificar que o tempo decorrido é superior ao intervalo de medição desejado, neste caso de 5 minutos, o contador de tempo decorrido é zerado e aciona-se os módulos para novas aferições de temperatura, umidade e som. Após as coletas, os dados são anexados ao arquivo já existente da data atual ou armazenados em um novo arquivo caso seja a primeira aferição do dia. Com isso, ao receber o sinal via bluetooth

do aparelho conectado (nesse caso o caractere 'g'), os arquivos salvos são consultados para recuperar os dados dos últimos 7 dias e enviá-los, através de um *buffer*, para o aparelho conectado com o módulo bluetooth, caso haja. Por fim, é enviada uma mensagem que indica o fim de um arquivo de dados, com o conteúdo "@".

4.2.3 Arquivos auxiliares

A fim de manter uma melhor organização nos arquivos do código, foram criados os arquivos *files.cpp* e *files.h*, que encapsulam os métodos responsáveis pela manipulação de arquivos, tal como ler e gravar um arquivo do cartão de memória e escrever dados em um arquivo já existente.

4.2.4 Bibliotecas externas

As bibliotecas externas utilizadas neste projeto foram:

- Adafruit_BusIO
- Adafruit_Unified_Sensor
- DHT_sensor_library
- ESP32_BleSerial
- RTClib

As pastas com os conteúdos de cada uma podem ser encontradas no repositório do microcontrolador no GitHub[6] na pasta *libraries*.

4.3 Estrutura do Circuito

O circuito tem como elemento central o *ESP32*, pequeno microcontrolador com capacidade de comunicação sem fio, através de *Wi-Fi* ou *Bluetooth*. É a partir dele e de informações passadas à ele que todos os módulos serão controlados. O circuito também é composto por 3 módulos de sensores, o *Sensor de Som* e os *Sensores de Temperatura e Umidade DHT11*, que serão os dispositivos responsáveis por coletar os dados da colméia. Por fim, o circuito também possui um módulo *RTC DS3231* (função de captar o dia e hora em que os dados são coletados) e um módulo *SD Card* (função de armazenamento dos dados coletados). Todos os módulos, bem como o microcontrolador, são conectados conforme a Figura 3 (Pin Out também pode ser encontrado no repositório[6]).

4.4 Limitações de Hardware

Devido a fatores como disponibilidade de redes de Wi-Fi, consumo de energia elétrica constante e complexidade de operação com tecnologias LoRa, o desenvolvimento do projeto foi limitado a um escopo menor do que originalmente desejado. As limitações tornam necessárias a utilização de bluetooth para a coleta de dados e uma fonte de energia elétrica

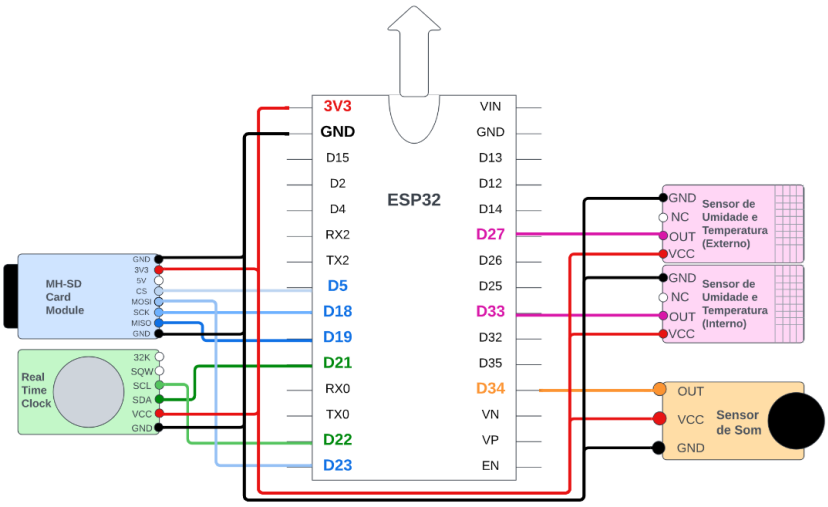

Figura 3: PinOut
Fonte: Elaborado pelos Autores (Plataforma LucidCharts)

permanente, ao invés de um sistema de captação de energia solar, por exemplo. Outra limitação presente foi a baixa disponibilidade e consequente alto preço de equipamentos no mercado, que impediu a utilização de outras tecnologias mais poderosas, como um Raspberry Pi ou similares.

Outras limitações também se apresentam na imprecisão dos instrumentos. Os sensores de umidade e temperatura, por exemplo, possuem um grau de incerteza relativamente alto, o que dificulta o processo de comparação entre as medições internas e externas. O microfone, por sua vez, também apresenta um problema para obter o nível de ruído em dB, uma vez que uma conversão lógica entre o valor de tensão lido e a aproximação em dB leva a mais incertezas. O relógio digital também apresentou problemas, onde o mesmo acaba perdendo o horário correto por alguns minutos após alguns dias de operação. Por fim, o bluetooth de baixa energia (BLE), em conjunto com o tamanho limitado de memória do ESP, causam uma perda de informação na transferência. A necessidade de utilizar BLE ao invés do bluetooth tradicional se deve ao fato de uma quantidade considerável de smartphones modernos não suportarem mais o bluetooth clássico, isso se reflete também na falta de bibliotecas e ferramentas de desenvolvimento para tal tecnologia.

5 Implementação Aplicação Mobile

5.1 Fluxo Geral

A aplicação móvel é um projeto criado com o framework *Flutter*[7], criado pelo Google. A partir dele, é possível construir aplicativos nativos para as plataformas Android e IOS. No projeto, a plataforma principal será o Android. Na Figura 4 é possível visualizar o funcionamento do framework dividido por camadas.

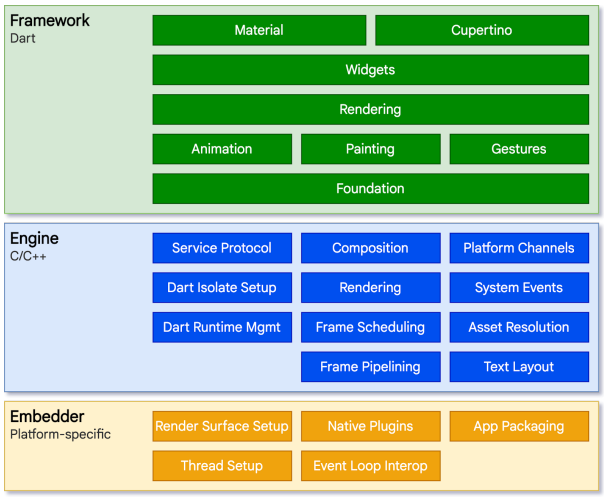

Figura 4: Camadas Framework
Fonte: Documentação Flutter[8]

Todo código pode ser encontrado no repositório da Aplicação Mobile no GitHub[9]. Os objetivos da aplicação são:

- Comunicação Bluetooth com o microcontrolador
- Armazenamento e análise de dados
- Gerenciamento de estados dos dados coletados
- Apresentar formas de análise dos dados

Os próximos tópicos abordam a construção do aplicativo em cada um dos aspectos mais importantes para atingir os objetivos necessários.

5.2 Arquivo Inicial

Toda aplicação flutter possui um arquivo chamado *main.dart* localizado na raíz do diretório *lib* do projeto. Esse arquivo possui a função *main()*, a qual é o ponto de início de toda aplicação. Para que o código dart processe o flutter, uma função *runApp()* precisa ser chamada com o *Widget* inicial do projeto.

No começo da *main*, antes de iniciar as regras do projeto de fato, é necessário chamar um método de nome *WidgetsFlutterBinding.ensureInitialized*, o qual garante que a camada de *widgets* do flutter seja iniciada. Posteriormente, são instanciadas todas as classes relacionadas ao módulo bluetooth, este que será explorado no item 5.6. Finalmente, é chamada a função *runApp()*. Neste projeto, a função *runApp* possui toda configuração dos *providers*, que são os gerenciadores de estados da aplicação, tópico que será abordado com mais clareza no item 5.9. Por fim, a função *runApp* renderiza o *Widget MyApp*, o qual possui toda a composição da aplicação flutter.

5.3 Models

- *ChartItem*: presente no caminho *bee_monitoring_app/Commons/Models/ChartItem* e consumida pelo manipulador presente em *bee_monitoring_app/Scenes/Chart/Handlers/.../DataHandler*, o modelo é responsável por representar um ponto no gráfico, com as propriedades *date*, *temperatureInside*, *temperatureOutside*, *humidityInside*, *humidityOutside* e *sound*. Sendo *date* a propriedade utilizada no eixo x e as demais no eixo y, apresentadas de acordo com a seleção do usuário de quais são as propriedades desejadas para exibição.

- *Item*: presente no caminho *bee_monitoring_app/Commons/Models/Item* e consumida pelo serviço presente em *bee_monitoring_app/Commons/Service*, o modelo é responsável por representar um dado de coleta realizada, ou seja, é utilizado no parse dos dados coletados presentes no formato *json*. Tal objeto é consumido pelas cenas para popular os elementos que representam as informações.

5.4 Telas

5.4.1 Resumo das coletas

Nesta tela (Figura 5) são apresentadas as médias, mínimas e máximas capturadas em cada um dos sensores presentes na colméia. Também está presente o menu do app, com possibilidade de realização de nova coleta, seleção da coleta que o usuário deseja consultar e exportação de dados em um arquivo *.json*.

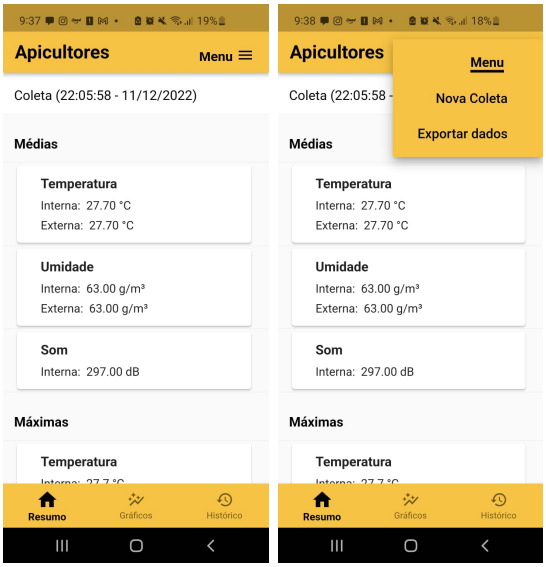

Figura 5: Resumo das coletas
Fonte: Elaborado pelos Autores

5.4.2 Dashboard para análise das coletas

Nesta tela (Figura 6) são apresentados os dados coletados através de um gráfico de linhas, no qual é possível sobrepor as diferentes informações. Foi desenvolvido o componente de opções para seleção do modo de visualização, com média diária ou coleta individual. Também é possível selecionar o dia em que deseja-se observar os dados. A funcionalidade também suporta paginação, para que seja possível navegação entre os dias referentes a coleta.

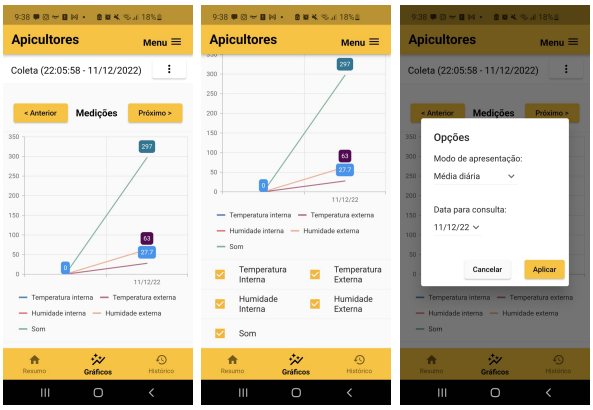

Figura 6: Dashboard para análise das coletas
Fonte: Elaborado pelos Autores

5.4.3 Histórico de coletas

Nesta tela (Figura 7) é apresentado o histórico de coletas individuais, seccionado por propriedade coletada.

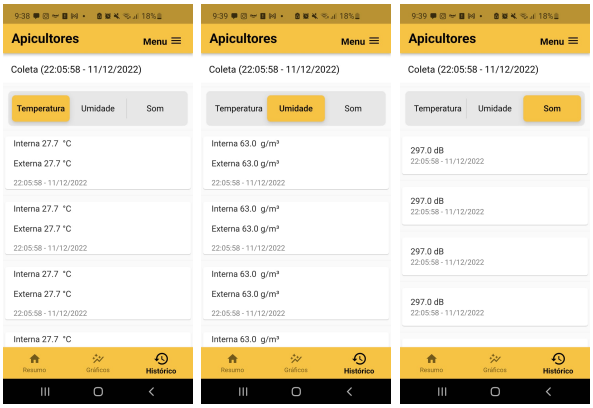

Figura 7: Histórico de coletas
Fonte: Elaborado pelos Autores

5.5 Definições técnicas da interface do usuário

Para o desenvolvimento das cenas, utilizou-se a arquitetura MVVM, isto é, separando-as nas camadas *view*, *viewModel* e *model*, com *factorys* para os *widgets* serem construídos individualmente. As cenas estão presentes no caminho *bee_monitoring_app/Scenes/*, sendo elas *Chart*, *Home* e *Timeline*. Cada *tab bar* foi considerada como uma cena com seus próprios controladores, feita de maneira independente.

Além disso, criou-se um controlador para navegação presente em *bee_monitoring_app/... /Scenes/Navigation* e um grupo de classes comuns consumidas por toda a aplicação, presente em *bee_monitoring_app/Commons/*.

As classes comuns foram separadas em *enums*, *modelos* e *serviço*, tais arquivos são responsáveis pelos cálculos e consumo dos dados.

Para o desenvolvimento das telas, foram utilizadas as seguintes dependências:

- *cupertino_icons*: Utilizada para implementação do Material design, com ícones padrão do sistema Android.
- *syncfusion_flutter_charts*: Utilizada para implementações relacionadas ao gráfico, com componentes prontos para exibição de gráfico dinâmicos.
- *syncfusion_flutter_gauges*: Utilizada para implementações relacionadas ao gráfico, possibilitando atualização de dados com animação e UI mais bonitas.
- *flutter_screenutil*: Utilizada para suportar fonte dinâmica no projeto, deixando-o mais acessível.

Para definição dos elementos de UI foram utilizados os elementos padrão do Android, para facilitar o entendimento do usuário com as ferramentas disponibilizadas na aplicação, bem como a manutenção do código realizado, por serem elementos básicos da programação em flutter.

5.6 Camada Bluetooth

Pode-se dizer que a conexão bluetooth é o módulo principal do aplicativo, pois é através dela que os dados são recebidos no aparelho, analisados e dispostos em gráficos para validação do usuário.

O padrão a ser utilizado na aplicação é o BLE[10] (*Bluetooth Low Energy*), porque é uma versão de baixa potência do Bluetooth destinada a sensores e acessórios de baixa potência. É ideal para aplicações que não exigem conexão contínua, mas dependem da longa duração da bateria. Dessa forma, o uso deste para a transferência de dados do hardware para o aparelho se torna uma boa alternativa. Este módulo foi desenvolvido utilizando uma biblioteca pública mantida pela empresa Philips Hue, a *flutter_reactive_ble*[11], que possui métodos facilitadores para escanear dispositivos; estabelecer uma conexão; manter conexões com múltiplos dispositivos, tópico no qual não foi abordado neste projeto porém com facilidade para adaptação; ler características e inscrever-se em uma característica. Estes métodos foram utilizados como dependências para as instâncias do serviço de comunicação

BLE construído no projeto. O módulo está no diretório *lib/Commons/ble(∼)*, onde estão contidas 5 classes desenvolvidas, incluindo uma classe útil destinada aos logs do serviço. São elas:

- ReactiveState
- BleStatusMonitor
- BleScanner
- BleLogger
- BleDeviceInteractor
- BleDeviceConnector

Todas as classes referentes ao serviço bluetooth são trabalhadas com injeção de dependência, onde todas as dependências são injetadas na função *main* do arquivo *main.dart*. Dessa forma, foi possível desacoplar a dependência de uma biblioteca externa, e o projeto depende apenas das abstrações criadas. As classes são instanciadas e assim disponibilizadas para o restante da aplicação para seu uso a partir de *providers*, este que será mais explorado no tópico de gerenciamento de estados.

A primeira classe a ser detalhada é a *BleScanner* no arquivo *∼/ble_scanner.dart*. Esta contém dois métodos necessários: *startScan* e *stopScan* que são chamados na tela de descoberta de dispositivos bluetooth.

O método de ativação da busca de aparelhos, quando chamado, utiliza das funções presentes na biblioteca e cria uma inscrição no serviço de descoberta, disponibilizado pela biblioteca externa. Desta forma, todo aparelho encontrado é indexado e a seguir, o estado de dispositivos encontrados, chamado de *discoveredDevices* é distribuído para o restante da aplicação, e atualizado com os aparelhos encontrados.

O método de parada do scanner apenas cancela a inscrição criada no método de escaneamento e atualiza o status da inscrição no *provider*.

A próxima classe a ser discutida chama-se *BleDeviceConnector*. Ela trata da conexão e desconexão com os aparelhos encontrados pelo serviço de escaneamento utilizando os respectivos métodos principais: *connect* e *disconnect*.

O *connect* possui como parâmetro o *deviceId* do aparelho terceiro com o qual se estabelecerá uma conexão. Utiliza-se o método de conexão da biblioteca, a qual devolve uma *Stream<ConnectionStateUpdate>*. O método então ouve todas as mudanças e atualiza o estado da conexão nos *providers*, garantindo que todo o app tenha informação a respeito da conexão.

Semelhante ao método *stopScan* da classe de escaneamento, o *disconnect* cancela a conexão existente e atualiza o estado no *provider*.

Com o aparelho conectado, no caso deste projeto o microcontrolador, precisa-se interagir com os serviços e características provenientes do mesmo para garantir o recebimento dos dados. Esta responsabilidade é implementada na classe *BleDeviceInteractor*, encontrada no diretório do módulo.

A transferência dos dados se dá em 2 passos:

1. É realizada uma inscrição na característica disponível com o id "6e400001-b5a3-f393-e0a9-e50e24dcca9e" utilizando um método *_subScribeToCharacteristic*. Dessa forma fica estabelecido um buffer de dados: quando um dado for transferido a partir do microcontrolador o aplicativo ficará escutando todas as mudanças, essas recebidas neste buffer como um *Stream<List<int>>*, que serão salvas diretamente no arquivo.

2. Para que os dados comecem a ser trafegados, o aplicativo precisa enviar um valor na característica, neste caso o caractere 'g', para que o microcontrolador inicie a transferência. Para isso é utilizado o método *_writeWithResponse* com o valor de parâmetro (106) (valor em inteiros do caractere 'g').

A classe de interação possui outros métodos que são implementações dos outros tipos de operações disponíveis na biblioteca *FlutterReactiveBle* , são eles: *_bleDiscoverServices*, *_readCharacteristic*, *_writeWithResponse*, *_writeWithoutResponse*, *_subScribeToCharacteristic*.

5.7 Camada de arquivos

O aplicativo, ao receber os dados a partir da comunicação bluetooth, precisa salvá-los em algum lugar para que estes estejam disponíveis para análise do usuário, mesmo longe do microcontrolador.

Para armazenar estas informações, o flutter disponibiliza algumas alternativas, são elas: utilização do *file system* do próprio aparelho, utilização de pacotes terceiros como o *Hive*, um bancos de dados *NO-SQL*.

Foi definido o uso de leituras e escritas de arquivos json diretamente no *file system*, levando em consideração a variação no tamanho que os dados podem ter e o tempo necessário para salvar os dados. Para garantir que todos os dados são salvos, todo conteúdo recebido do buffer de entrada de dados é convertido em uma string e escrito no arquivo local, chamado de *data.json*. Dessa forma, os dados não são pré-processados, o que garante uma agilidade na leitura e escrita dos mesmos. Vale ressaltar que, a cada nova sincronização de dados com o microcontrolador, os dados antigos serão sobrescritos com os novos dados e salvos no arquivo *data.json*.

A classe responsável por implementar esse serviço de arquivos é chamada de *FileManager*, contida em *lib/common/services/file_manager.dart*. Esta classe é implementada seguindo o padrão de *singletons*, para que não haja a possibilidade de mais de uma instância do arquivo estar aberta ao mesmo tempo, o que garante que não ocorreram operações concorrentes no mesmo arquivo.

Inicialmente são declarados dois getters para as propriedades *_directoryPath* e *_jsonFile*, que possuem as respectivas funções:

- Utilizar da biblioteca IO para detectar o diretório interno do aplicativo no aparelho do usuário e retornar seu caminho, desta forma é possível utilizar o caminho necessário para cada plataforma, no caso deste, Android.

- Abrir uma instância do arquivo *data.json* utilizando seu caminho no diretório local, que conterá todos os dados recebidos pelo bluetooth, utilizando da propriedade *_directoryPath*.

Como visto na seção de comunicação bluetooth, os dados serão recebidos como um tipo *Stream<List<int>>*, o que possibilita uma melhora na performance de escrita de dados no arquivo, se gravarmos os dados sequencialmente. Ou seja, conforme um dado chega pelo serviço bluetooth, ele será automaticamente salvo no arquivo local.

Essa função foi implementada pelo método *writeJsonFile*, que recebe como parâmetro um *List<Int>subscribeStreamValue*. Este método primeiramente acessa uma instância do arquivo local para sua posterior escrita. O parâmetro *subscribeStreamValue* é transformado em uma string e o resultado é anexado ao final do arquivo existente.

Outro método deste serviço trata da leitura dos dados contidos no arquivo local e é chamado de *readJsonFile*, o qual utiliza da instância do arquivo local e verifica sua existência. Caso exista, é realizada a leitura do conteúdo do arquivo e esse conteúdo é repassado ao serviço de análise de dados que transformará o conteúdo do arquivo em dados do tipo *List<Item>*.

O último método deste serviço é o *exportJson*, com a função de localizar e exportar o arquivo contendo os dados da última coleta. Este método utiliza a biblioteca *share_plus*[12] para realizar a exportação e compartilhamento do arquivo. A funcionalidade foi pensada para contrapor o problema de sobrescrição dos dados antigos pelos novos dados coletados, fazendo com que o usuário seja capaz de exportar estes dados antes de realizar uma nova coleta, evitando a perda de dados.

5.8 Camada de análise

A análise de dados é realizada pelo serviço *FilesParser* localizado no arquivo de caminho: *lib/common/services/file_parser.dart*. A análise é responsável por receber o conteúdo do arquivo e transformá-lo em uma lista de objetos que será repassada para a aplicação, assim tornando possível a construção dos gráficos.

A classe necessita do conteúdo do arquivo no construtor da classe como uma string. Assim é chamado o método *_decodeAndParseJson*,o qual utiliza do método construtor do item a partir de um json. O retorno dessa função é uma lista de itens.

```
Future<List<Item>> _decodeAndParseJson() async {
    try {
        final jsonData = jsonDecode(encodedJson);
        final resultsJson = jsonData['data'] as List<dynamic>;
        List<Item> results = [];
        for (var item in resultsJson) {
            results.add(Item(
            item!['ti'].toString(),
            item!['te'].toString(),
            item!['ui'].toString(),
            item!['ue'].toString(),
            item!['s'].toString(),
            DateFormat("yyyy-MM-dd hh:mm:ss").parse(item!['ts']),),);
        } return results;
    } catch (e) {
        print(e);
        rethrow;
}   }
```

5.9 Gerenciamento de Dados

Com os dados da colmeia já armazenados em um arquivo local, e com o serviço de leitura, é preciso disponibilizar esses dados para o restante da aplicação. Para isso foi necessário decidir e configurar algum método de gerenciamento de estados. Para o flutter, existem algumas abordagens mais comuns utilizadas pela comunidade de desenvolvimento, como por exemplo: *Provider*, *BloC*, *Redux*.

Neste projeto foi utilizado o *Provider* para o gerenciamento de estados, com base em um uso amplo das bibliotecas pela comunidade e facilidade em encontrar artigos sobre sua utilização. Este pacote trabalha com o conceito de singletons, ou seja, uma classe que conterá sempre uma única instância. Assim, essa instância da classe será a mesma em todas as camadas da aplicação, e, dessa forma, terá um único valor compartilhado na aplicação.

Para utilizar a biblioteca *Provider*, é necessário estender a classe *ChangeNotifier* cuja responsabilidade é notificar os widgets que estão consumindo a informação caso ocorra alguma alteração. Dessa forma, foi criada uma classe *JsonRepository* como no código abaixo. Neste caso a informação é uma propriedade da classe chamada de *_items*, o qual possui um *getter*. A classe também possui um método assíncrono chamado *readData* que é responsável pelo carregamento dos dados do arquivo local na propriedade *_items*.

Vale ressaltar que este método utiliza uma instância da classe *FileManager* para a leitura e análise dos dados contidos no arquivo. No final dessa execução, há uma chamada para o método *notifyListeners*, presente na classe *ChangeNotifier*. Dessa forma, todos os consumidores desses dados serão notificados.

```
class JsonRepository extends ChangeNotifier {
    List<Item> _items = [];
    List<Item> get items => _items;
    readData() async {
        final result = await FileManager().readJsonFile();
        if (result != null) {
            _items = result;
        }
    notifyListeners();
    }
}
```

Com um repositório criado, é preciso configurar como os Widgets consumirão os dados. Utilizando o Provider, é necessário adicionar na função *runApp* a função *Multiprovider*, que recebe uma lista com todos os providers necessários para nossa aplicação. Note que cada provider possui um tipo e valor. São eles:

- *ChangeNotifier*: Este tipo torna a leitura dos dados reativa. Dessa forma, quando houver mudança nos dados do *JsonRepository*, os Widgets dependentes irão atualizar.

- *Provider.value*: Estes dados são valores necessários utilizados pelo serviço bluetooth da aplicação.
- *StreamProvider*: Este é um tipo para retornar dados do tipo Stream no flutter. No app é usado para monitorar os dispositivos bluetooth que estão próximos e os estados de uma conexão bluetooth.

Desta forma, qualquer widget utilizado dentro de *MyApp* poderá acessar e consumir os dados de qualquer provider necessário.

A última modificação necessária é assegurar que os dados já tenham sido lidos do arquivo local, caso exista, durante o primeiro carregamento do aplicativo. Para isso é necessário atualizar o *Widget MyApp*. Como é visto no código abaixo, utiliza-se o contexto fornecido para acessar o repositório e executar o método *readData*, garantindo as informações necessárias.

```
class MyApp extends StatelessWidget {
    const MyApp({super.key});
    @override
    Widget build(BuildContext context) {
        context.read<JsonRepository>().readData();
        ...
```

6 Resultados

O vídeo com a demonstração da plataforma mobile pode ser encontrado na referência[13] com o nome "Demonstração Aplicativo Mobile". No vídeo, o fluxo apresentado é do aplicativo sendo aberto pela primeira vez e realizando sua primeira coleta de dados. A varredura de dispositivos próximos pelo bluetooth é feita quase que instantaneamente e não apresenta gargalos, assim como a chamada de conexão ao dispositivo. Foram encontrados alguns problemas de performance somente durante a transferência de dados pelo bluetooth. Por ser um grande fluxo de dados que está sendo recebido e salvo logo em seguida, o aplicativo apresenta quedas de frames e pequenos travamentos. Na mesma pasta de arquivos[13] é possível encontrar registros de uma simulação da instalação do projeto em uma colméia.

A coleta de dados leva em torno de 6 segundos para ser finalizada (com um *timeout* de 20 segundos), e, nesse tempo, há uma transferência de cerca de 2.016 objetos de dados, com uma perda de aproximadamente 15% dos dados por coleta. A partir dos testes realizados, conclui-se que os dados são enviados pelo microcontrolador, porém são perdidos na fase de transmissão e de recebimento dos dados no lado do aplicativo. Sendo assim, medidas como a simplificação dos nomes dos atributos e o envio de um objeto completo por stream de dados foram implementadas. Estas medidas servem para que não ocorra a perda de fragmentos dos dados, evitando assim que o arquivo json final fique corrompido. O que ocorre então são perdas de coletas completas, mas nunca mais de duas de forma consecutiva, levando assim a um máximo de 15 minutos sem dados coletados (intervalo de 5 minutos entre cada coleta).

O aplicativo final ficou com um tamanho de 20MB e pode ser baixado pelo repositório[9].

7 Documentação

7.1 Montagem

7.1.1 Equipamento

Para a montagem do circuito são necessários:

- 1 Placa perfurada ilhada
- 1 Módulo microcontrolador ESP32
- 1 Módulo sensor de som
- 2 Módulos sensor de temperatura e umidade DHT11
- 1 Módulo RTC DS3231
- 1 Módulo SD Card
- 1 Cartão SD
- 1 Fonte DC 5V 3A micro-USB
- 1 Cabo micro-USB
- Soquetes (diferentes tamanhos)
- Fios de cabo de rede par trançado (diferentes tamanhos)
- Jumpers Macho/Macho e Macho/Fêmea
- Estanho e ferro de solda

7.1.2 Circuito

O primeiro passo para a montagem do circuito é dispor os módulos por cima da placa para visualizar como o encaixe será feito. Depois disso, é necessário encaixar os soquetes (tamanho compatível com cada módulo) e soldá-los na placa. *Importante*: Soldar apenas o soquete na placa e encaixar os módulos depois para não correr o risco de queimar nenhum equipamento.

Com os soquetes soldados à placa, é a vez de fazer todas as ligações necessárias com os fios de cabo rede (desencapados) ou jumpers, de acordo com a preferência, e soldá-los na placa. O Pinout das ligações foi mencionado na seção 4.3 e também pode ser encontrado no repositório do Microcontrolador no GitHub[6].

7.1.3 Instalação

Com a placa montada, é necessário realizar uma instalação do código do arduino na placa para que ela funcione. Para isso, seguir os passos:

1. Baixar e instalar Arduino IDE[14] na versão de preferência (recomendado versão mais nova). Verificar se possui instalação do driver usb, que é necessário para a IDE identificar a placa.
2. Abrir no Arduino IDE o projeto encontrado no repositório[6].
3. Adicionar a pasta libraries[6] em sketchbook/libraries no local em que a IDE foi instalada.
4. Conectar a placa ao computador com o cabo micro-USB.
5. Na IDE, selecionar a placa ESP32 Dev Module e a porta na qual a placa está conectada.
6. Clicar em upload na IDE e segurar o botão Boot na placa até que inicie a porcentagem de envio no terminal da IDE.

7.2 Execução

Com a montagem finalizada, basta conectar a placa à fonte com o cabo micro-USB e baixar o executável da aplicação mobile (disponível no repositório da Aplicação Mobile no GitHub[9]). Após isso, o sistema está pronto para o uso.

8 Conclusão e Trabalhos Futuros

Apesar de uma proposta inicial envolvendo um sistema IoT, limitações físicas como falta de conexão com internet motivaram uma mudança para um sistema capaz de funcionar sem conexão com a internet. A solução escolhida foi um aplicativo capaz de se comunicar com o microcontrolador através de bluetooth. Esse meio de comunicação, apesar de suas limitações como perda eventual de dados, se mostrou satisfatório para as necessidades apresentadas pelo projeto.

No aplicativo, alguns pontos de melhoria se destacam. Primeiro, na versão desenvolvida neste projeto só é possível analisar os dados referentes a apenas 1 colméia. Logo, expandir essa análise para múltiplas colméias torna-se um importante passo para um trabalho futuro. Outro ponto importante é o armazenamento de arquivos no aplicativo. Nesta versão, toda sincronização nova com o hardware apaga os dados já salvos, portanto criar um mecanismo de sistema de arquivos mais robusto também é um ponto de melhoria, assim como uma posterior armazenagem de dados em algum serviço web, como por exemplo o Firebase, o que tornaria a aplicação mais versátil e mais leve. Uma solução mais simples seria implementar a funcionalidade de importação de arquivos json como forma de coleta. Com isso, as coletas antigas poderiam ser exportadas e posteriormente importadas para serem analisadas novamente.

Com o avanço das funcionalidades e um aumento na quantidade de dados, a análise de dados pode se tornar um ponto de gargalo no aplicativo. Para minimizar este efeito, é possível utilizar Isolates[15] para o processamento de dados, garantindo que o trabalho seja executado de forma isolada e simultânea, sem afetar a interface do usuário.

Referências

[1] A. Gillis, *What is internet of things (IoT)?*. 2021. Disponível em: https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

[2] *Assentamentos*. Governo Federal, 2020. Disponível em: https://www.gov.br/incra/pt-br/assuntos/reforma-agraria/assentamentos.

[3] *ESP32: Guide for MicroSD Card Module using Arduino IDE*. Random Nerd Tutorials, 2021. Disponível em: https://randomnerdtutorials.com/esp32-microsd-card-arduino/.

[4] *ESP32 with DHT11/DHT22 Temperature and Humidity Sensor using Arduino IDE*. Random Nerd Tutorials, 2021. Disponível em: https://randomnerdtutorials.com/esp32-dht11-dht22-temperature-humidity-sensor-arduino-ide/.

[5] *ESP32 and DS3231 RTC example*. ESP32 Learning, 2017. Disponível em: http://www.esp32learning.com/code/esp32-and-ds3231-rtc-example.php.

[6] *Microcontroller*. GitHub, 2022. Disponível em: https://github.com/Apicultores/microcontroller.

[7] *Flutter Documentation*. Disponível em: https://docs.flutter.dev/.

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

[9] *Mobile App*. GitHub, 2022. Disponível em: https://github.com/Apicultores/mobile-appr.

[10] M. Ziemla, *Bluetooth Low Energy in Flutter – An Overview*. Leancode, 2022. Disponível em: https://leancode.co/blog/bluetooth-low-energy-in-flutter.

[11] *Flutter reactive BLE library*. Pub.dev, 2022. Disponível em: https://pub.dev/packages/flutter$_r eactive_b le$.

[12] *Share plugin*. Pub.dev, 2022. Disponível em: https://pub.dev/packages/share$_p lus$.

[13] *Pasta contendo registros dos resultados*. Google Drive, 2022. Disponível em: https://drive.google.com/drive/folders/1lt0B2CqWQ3IE705Nnd22V3n4Sf5IlWAe.

[14] *Arduino IDE*. Disponível em: https://www.arduino.cc/en/software

[15] J. Essien, *Flutter isolates – everything you need to know*. Codemagic, 2022. Disponível em: https://blog.codemagic.io/understanding-flutter-isolates/.