INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO AMAZONAS
CAMPUS MANAUS-DISTRITO INDUSTRIAL
ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Rodrigo Pereira Silva

Desenvolvimento de dispositivo para telemetria e monitoramento remoto de florestas

Manaus – AM

2025

Rodrigo Pereira Silva

Desenvolvimento de dispositivo para telemetria e monitoramento remoto de florestas

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia de Controle e Automação do Campus Manaus Distrito Industrial, Instituto Federal de Educação, Ciência e Tecnologia do Amazonas (CMDI/IFAM), como requisito parcial para obtenção do grau de Bacharel.

Orientador: Prof. Dr. Fernando Rodrigues de Almeida.
Coorientador: Dr. Edevaldo Albuquerque Fialho

Manaus-AM

2025

Dados Internacionais de Catalogação na Publicação (CIP)

Silva, Rodrigo Pereira.
S586d Desenvolvimento de dispositivo para telemetria e monitoramento remoto de florestas / Rodrigo Pereira Silva. — Manaus, 2025.
62f.: il. color.

Monografia (Graduação) — Instituto Federal de Educação, Ciência e Tecnologia do Amazonas, *Campus* Manaus Distrito Industrial, Curso de Engenharia de Controle e Automação, 2025.
Orientador: Prof. Fernando Rodrigues de Almeida, Dr.
Coorientador: Edevaldo Albuquerque Fialho Dr.

1. Telemetria remota. 2. IoT. 3. LoRa. I. Almeida, Fernando Rodrigues. II. Fialho, Edevaldo Albuquerque. III. Instituto Federal de Educação, Ciência e Tecnologia do Amazonas. IV. Título.

CDD 629.89

Elaborada por Oziane Romualdo de Souza (CRB11/ nº 734).

ANEXO 7

ATA DE DEFESA PÚBLICA DO TRABALHO DE CONCLUSÃO DE CURSO

Aos 07 dias do mês de abril, de 2025, às 15:00 h, o(a) discente Rodrigo Pereira Silva apresentou o seu Trabalho de Conclusão de Curso para avaliação da Banca Examinadora constituída pelos seguintes integrantes: Prof(a). Fernando Rodrigues de Almeida (docente-orientador), Prof(a). Edevaldo Albuquerque Fialho (coorientador), Prof(a). Renan Cavalcante Santos (Membro 1) e Prof(a). Ricardo Augusto Medeiros de Oliveira (Membro 2). A sessão publica de defesa foi aberta pelo(a) presidente da banca, que apresentou a Banca Examinadora e deu continuidade aos trabalhos, fazendo uma breve referência ao TCC, que tem como título Desenvolvimento de dispositivo para telemetria e monitoramento remoto de florestas. Na sequencia, o(a) discente teve até 30 minutos para a comunicação oral de seu trabalho. Cada integrante da banca examinadora fez suas arguições após a defesa do mesmo. Ouvidas as explicações do(a) discente, a banca examinadora, reunida em caráter sigiloso, para proceder à avaliação final, deliberou e decidiu pela APROVAÇÃO com média final 9,4 (NOVE VÍRGULA QUATRO) do referido trabalho.

Foi dada ciência ao(à) discente que a versão final do trabalho deverá ser entregue até o dia 15 / 04 / 25, com as devidas alterações sugeridas pela banca. Nada mais havendo a tratar, a sessão foi encerrada às 16 h 00 min, sendo lavrada a presente ata, que, uma vez aprovada, foi assinada por todos os membros da Banca Examinadora e pelo(a) discente.

Prof.(a) Orientador(a)/Presidente: _____

Prof.(a) Avaliador 1: _____

Prof.(a) Avaliador 2: Renan Cavalcante Santos

Discente: Rodrigo Pereira Silva

CO-ORIENTADOR: _____

Dedicated to those who perished on the climb.

RESUMO

O avanço das tecnologias de Internet das Coisas (IoT) possibilita novas abordagens para o monitoramento ambiental, especialmente em áreas remotas e de difícil acesso, como a Floresta Amazônica. A necessidade de acompanhar mudanças ambientais, prevenir desmatamento e incêndios florestais, bem como facilitar pesquisas científicas, motiva o desenvolvimento de soluções tecnológicas autônomas para a coleta e transmissão de dados. Neste contexto, este trabalho propõe o desenvolvimento de um sistema de telemetria baseado em IoT para monitoramento remoto de florestas, utilizando a tecnologia LoRa para comunicação em uma rede em malha. O sistema é composto por dois dispositivos principais: o Nó de Monitoramento, responsável pela medição de parâmetros ambientais como temperatura, umidade, pressão atmosférica e compostos orgânicos voláteis; e a Ponte de Comunicação, encarregada de retransmitir os dados para uma central de monitoramento via protocolo MQTT. Os testes laboratoriais e de campo demonstraram a eficiência da solução na transmissão de dados. Os resultados indicam que a abordagem desenvolvida é viável para aplicações de monitoramento ambiental, podendo ser aprimorada com otimizações no consumo de energia e na estrutura física dos dispositivos.

Palavras-chave: IoT; LoRa; Monitoramento Florestal; Telemetria Remota.

ABSTRACT

The advancement of Internet of Things (IoT) technologies enables new approaches to environmental monitoring, especially in remote and hard-to-reach areas such as the Amazon Rainforest. The need to track environmental changes, prevent deforestation and forest fires, and facilitate scientific research drives the development of autonomous technological solutions for data collection and transmission. In this context, this work proposes the development of an IoT-based telemetry system for remote forest monitoring, utilizing LoRa technology for communication within a mesh network. The system consists of two main devices: the Monitoring Node, responsible for measuring environmental parameters such as temperature, humidity, atmospheric pressure, and volatile organic compounds; and the Communication Bridge, which transmits the data to a monitoring center via the MQTT protocol. Laboratory and field tests demonstrated the system's efficiency in data transmission. The results indicate that the proposed approach is viable for environmental monitoring applications and can be further improved with optimizations in energy consumption and the physical structure of the devices.

Keywords: IoT; LoRa; Forest Monitoring; Remote Telemetry.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADC	Analog-to-Digital Converter (Conversor Analógico-Digital)
API	Application Programming Interface (Interface de Programação de Aplicações)
CAD	Computer-Aided Design (Desenho Assistido por Computador)
CAM	Computer-Aided Manufacturing (Manufatura Assistida por Computador)
CMDI	Campus Manaus Distrito Industrial
CO_2	Dióxido de Carbono
CRC	Cyclic Redundancy Check (Verificação de Redundância Cíclica)
CSS	Chirp Spread Spectrum (Espectro Expandido por Chirp)
dB	Decibel
dBm	Decibel milliwatt
eCO_2	Equivalent Carbon Dioxide (Dióxido de Carbono Equivalente)
ESP IDF	Espressif IoT Development Framework
FDM	Fused Deposition Modeling (Modelagem por Deposição Fundida)
GHz	Gigahertz
hPa	Hectopascal
HTTP	HyperText Transfer Protocol (Protocolo de Transferência de Hipertexto)
I^2C	Inter-Integrated Circuit (Circuito Integrado Interconectado)
IDE	Integrated Development Environment (Ambiente de Desenvolvimento Integrado)
IFAM	Instituto Federal de Educação, Ciência e Tecnologia do Amazonas
IoT	Internet of Things (Internet das Coisas)
JSON	JavaScript Object Notation
LoRa	Long Range (Longo Alcance)
LoRaWAN	Long Range Wide Area Network (Rede de Longo Alcance de Área Ampla)
LPWAN	Low-Power Wide-Area Network (Rede de Baixo Consumo e Longo Alcance)
Mbit/s	Megabit por segundo
MHz	Megahertz
MQTT	Message Queuing Telemetry Transport
MP2.5 /MP10	Material Particulado 2.5µm / 10µm
OSI	Open Systems Interconnection (Interconexão de Sistemas Abertos)
PCI	Placa de Circuito Impresso

PLL	Phase-Locked Loop (Laço de Fase Travada)
PPB	Partes por Bilhão
PPM	Partes por Milhão
PWM	Pulse Width Modulation (Modulação por Largura de Pulso)
RAM	Random Access Memory (Memória de Acesso Aleatório)
RF	Radio Frequency (Frequência de Rádio)
RSI	Rotina de Serviço de Interrupção
RSSI	Received Signal Strength Indicator (Indicador de Intensidade do Sinal Recebido)
RTOS	Real-Time Operating System (Sistema Operacional de Tempo Real)
SCL	Serial Clock Line (Linha de Relógio Serial)
SNR	Signal-to-Noise Ratio (Relação Sinal-Ruído)
SPI	Serial Peripheral Interface (Interface Periférica Serial)
SS	Slave Select (Selecionador de Escravo)
DAS	Serial Data Line (Linha de Dados Serial)
SSID	Service Set Identifier (Identificador de Conjunto de Serviço)
TVOCs	Total Volatile Organic Compounds (Compostos Orgânicos Voláteis Totais)
UART	Universal Asynchronous Receiver-Transmitter (Transmissor-Receptor Assíncrono Universal)
USB	Universal Serial Bus
Wi-Fi	Wireless Fidelity

SUMÁRIO

1 INTRODUÇÃO

Telemetria de áreas remotas é uma área que atende diversas aplicações: exploração subaquática, florestal e até mesmo interplanetária. Essa mesma telemetria quando aplicada a áreas florestais amazônicas remotas e/ou de difícil acesso, através de dispositivos eletrônicos inteligentes que sejam autossustentáveis, abriria um leque de possibilidades de monitoramento.

Entretanto, coletar dados relacionados ao meio ambiente em partes profundas da floresta amazônica representa um desafio, devido ao grande esforço necessário para se locomover até o local, coletar os dados, ou deixar um sensor realizando as medições ao longo de um período pré-definido de tempo, e depois retornando para o local de instalação para reaver o dispositivo juntamente com os dados.

Considerando as dificuldades mencionadas acima, este estudo procura responder a seguinte questão: de que maneira poderíamos alterar o meio para adquirir esses dados de forma que facilite esta coleta, seja facilmente reprodutível e escalável?

Existem algumas alternativas de solução para esse problema: construir vias para facilitar a locomoção até os locais de interesse nos traria a comodidade do deslocamento, mas alteraria consideravelmente o meio, e não se prova como uma alternativa escalável.

Também seria possível construir uma rede de alimentação elétrica e de transmissão de dados cabeada para instalação dos dispositivos, uma alternativa que altera consideravelmente menos o meio se comparada com a anterior.

Entretanto a alternativa mais chamativa é a de desenvolver dispositivos “sustentáveis” capazes de gerar energia para sua própria operação, e transmissão remota dos dados coletados, o que atenderia os requisitos levantados de ser escalável e facilmente reprodutível.

O objetivo geral deste trabalho é desenvolver um dispositivo para realizar a telemetria de áreas remotas e de difícil acesso na Floresta Amazônica. Para isso, busca-se determinar os parâmetros físicos, químicos e biológicos de interesse para a coleta de dados, levantar as tecnologias aplicáveis à transmissão de informações no cenário amazônico, elaborar a arquitetura do sistema considerando a coleta, transmissão e apresentação dos dados, construir

o sistema de acordo com a arquitetura proposta e realizar testes de operação, e analisar os resultados obtidos para avaliar o desempenho da solução desenvolvida.

O desenvolvimento desse dispositivo facilitaria a execução de atividades de pesquisadores de campo, que necessitam de uma melhor maneira para adquirir esses dados, tornaria possível o acompanhamento do estado da qualidade do meio em que for instalado, assim como poderia ser aplicado na segurança florestal, detectando queimadas e, em uma aplicação futura, até mesmo ruídos de motosserra elétrica.

Alternativas para o monitoramento de florestas são escassas. No Brasil, o método mais empregado é o uso de satélites, através do INPE, para observação delas.

A implementação desse sistema, traria para o cenário da Floresta Amazônica a tecnologia de telemetria, que até então é empregada em sua grande maioria em centros urbanos, de distribuição elétrica e setor fabril.

2 REFERENCIAL TEÓRICO

Nessa seção são apresentados os conceitos necessários para desenvolvimento e compreensão do trabalho.

Segundo Almeida (2017), desmatamento, alteração de fauna e flora e alterações hídricas são alguns dos impactos ambientais comumente causados pelo ser humano. Uma das formas da geração desses impactos vem da urbanização não sustentável, do crescimento desordenado de cidades, e pelos efeitos secundários da grande migração de pessoas para atuação na área da indústria na Amazônia, a Zona Franca de Manaus.

Considerando o contexto de uma arquitetura IoT, que é uma rede de dispositivos físicos que estão conectados à internet, e são capazes de se comunicar uns com os outros (Vega-Rodriguez et al., 2019), surge a possibilidade do monitoramento remoto de florestas, incluindo a Amazônica, utilizando esse conjunto de tecnologias.

Ferreira et al. (2020) apresenta uma solução para localização de visitantes em unidades de conservação brasileiras utilizando LoRa, desenvolvendo uma estrutura que valida a operação dessa tecnologia de comunicação no bioma que é o foco do trabalho atual.

O trabalho de Sardar et al. (2018) mostrou que apesar da densidade da vegetação, chuva e outros fatores atrapalharem na transmissão, a tecnologia LoRa se sobressai em comparação a outras como Zigbee, WiFi e Bluetooth na transmissão de dados em florestas.

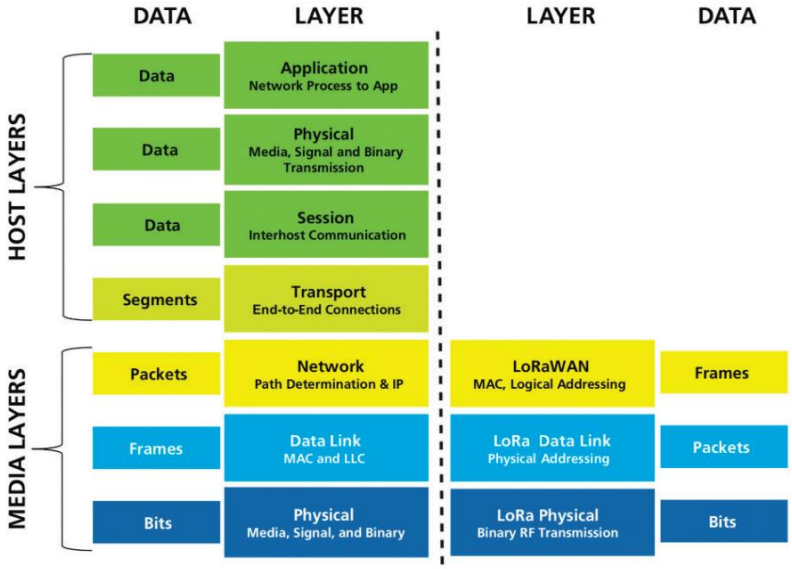
2.1 LoRa

É uma camada física de transmissão de dados de cunho proprietário, que faz parte da categoria de LPWAN. Utiliza de um derivado do *chirp spread spectrum* (CSS) para realizar essa transmissão, codificando os dados com bits de informação extra de tal maneira que reduza os ruídos e interferências, além de deixar esse parâmetro como configurável, podendo fazer um balanceamento entre taxa de dados e alcance (CENTENARO, 2016).

LoRa, do inglês *Long Range,* significa “longo alcance” e é exatamente isso que essa tecnologia se propõe a fazer: transmitir dados a longas distâncias, se comparadas com transceptores de mesma densidade tecnológica. LoRa por si

só, define a camada física e de enlace de dados no modelo *Open System Interconnection* (OSI), e por conta disso necessita de camadas superiores, como LoRaWAN, para realizar o envelopamento da comunicação (VAN EIJK, 2020). Na Figura 1 pode ser vista a presença de LoRa no modelo OSI.

Figura 1: LoRa no modelo OSI



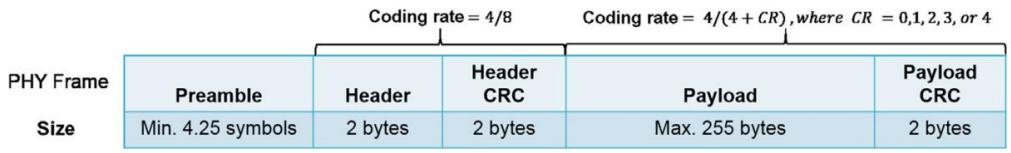
Fonte: VAN EIJK, 2020.

Dentro dessa tecnologia, existem alguns parâmetros que podem ser configurados para realização de controle fino da performance da rede estabelecida, além da frequência base de transmissão, outros parâmetros que mais afetam LoRa segundo Pham et al. (2020, p. 66) são o *spreading factor*, o *coding rate*, e a largura de banda. S*preading factor* significando “fator de espalhamento”, é um valor indicativo da relação entre a quantidade de símbolos e de *chirps*, quanto maior o fator de espalhamento, maiores são a quantidade de bits por símbolo, de duração do envio de cada símbolo e as chances de recebimento da transmissão. Seis fatores de espalhamento ortogonais são empregados em LoRa, 7 até 12. *Coding rate* significa taxa de codificação, e dita a quantidade de *forward error correction* (correção de erro antecipada) presente em um pacote transmitido via LoRa. Esse método de correção de erro envia bits redundantes e com informações para reconstrução da mensagem no caso de corrompimento durante transmissão e recebimento. Valores de *coding rate* vão

de 0 até 5, com 0 significando nenhuma codificação. Usar valores altos de fator de espalhamento e de taxa de codificação aumentam o tempo de transmissão dos pacotes. Usar valores maiores de largura de banda reduzem o tempo de transmissão, mas também diminuem a sensibilidade do receptor. Em LoRa diversos valores de largura de banda estão disponíveis, desde 7.8 até 500 kHz, sendo os valores de 125, 250 e 500 kHz os mais utilizados.

Após estabelecidos esses parâmetros de configuração, ainda faltam alguns parâmetros que devem estar de acordo tanto no transmissor quanto no receptor para realizar a transmissão de dados com sucesso. Na Figura 2 pode ser vista a estrutura completa de um pacote LoRa, contendo preâmbulo, cabeçalho, CRC do cabeçalho, *payload* e *payload CRC.* Preâmbulo é uma sequência de *chirps* com objetivo de sincronizar o receptor ao transmissor, e possui no mínimo 4,25 símbolos. O cabeçalho é opcional, sempre utiliza taxa de codificação 4, e contém informações sobre o *payload* e sobre o CRC do cabeçalho. O CRC do cabeçalho e CRC do payload servem para checar integridade do cabeçalho e do *payload* respectivamente. O *payload* contém os dados enviados pela aplicação que está utilizando LoRa (PHAM et al., 2020).

Figura 2: Estrutura de um pacote LoRa



Fonte: Pham et al, 2020.

2.2 PROTOCOLOS UTILIZANDO LORA

Nesta seção serão introduzidos alguns protocolos de comunicação disponíveis quando é utilizada a modulação LoRa como meio de transmissão.

2.2.1 LoRaWAN

Foi definido e é mantido pela LoRa Alliance®, e se trata de um protocolo de comunicação que ocupa a camada de rede no modelo OSI, como pode ser visto na Figura 1. É utilizado em *Low-Power-Wide-Area-Networks* (LPWANs)

como LoRa, e foi projetado para conectar “objetos” a redes regionais, nacionais ou globais, e tem como foco os requisitos da IoT como comunicação bidirecional, segurança de ponta a ponta, mobilidade e serviços de localização. A topologia obtida com esse protocolo é estrela. (LoRa Alliance, 2021).

2.2.2 LoRaMESH

Se trata da utilização da camada física LoRa para construção de uma rede com topologia em malha. Apesar de ter um nome, não há nenhum órgão regulador responsável por esse protocolo, assim a responsabilidade da definição e implementação dele fica a cargo do autor.

Nesse contexto, considerando redes com topologia em malha, SEMTECH (2015) diz que elas se beneficiam de redundância, aumentando as chances do envio com sucesso dos dados, já que cada nó da rede retransmite o dado recebido, assim, quando um dispositivo falha, o dado ainda é retransmitido por outro. Outro benefício é a maior facilidade de aumento de cobertura quando comparado com uma rede em estrela.

Apesar disso, esse tipo de rede também tem que lidar com alguns problemas únicos como o impacto no desempenho dela conforme o número de dispositivos aumenta, devido as constantes retransmissões de dados. Além disso, também consome mais energia que uma rede em estrela, já que todos os dispositivos no caminho até o nó final devem usar energia para transmitir o mesmo dado (SEMTECH, 2015).

2.3 SISTEMAS MICROCONTROLADOS

Principais características de sistemas microcontrolados e de protocolos utilizados para comunicação com periféricos e entre microcontroladores.

2.3.1 Conversor Analógico-Digital

A conversão de sinal analógico para digital é o processo de transformar a saída do circuito de amostragem e retenção em uma sequência de códigos binários que representam a intensidade do sinal de entrada analógico em cada momento de amostragem. Durante a amostragem e retenção, a amplitude do

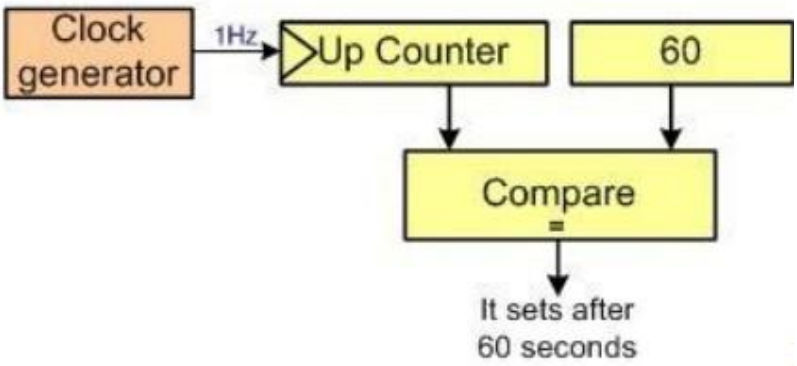
sinal analógico é mantida constante entre os ciclos de amostragem. Assim, a conversão para digital pode ser realizada utilizando um valor constante, em vez de um sinal analógico que muda durante o período de conversão, que é o intervalo entre as amostras. (FLOYD, 2006).

2.3.2 Temporizadores e contadores

Segundo Mazidi, Chen e Ghaemi (2018), quando surge a necessidade de realizar a contagem da ocorrência de eventos, como por exemplo o número de embalagens que passam numa esteira de uma fábrica, uma das soluções desse problema é a de conectar um sensor de presença a um contador de um microcontrolador. O contador assim pode ser definido como um periférico capaz de contabilizar eventos digitais genéricos.

Uma outro aplicação que utiliza contadores é para iniciar e terminar uma tarefa após um período desejado passar. Nessa aplicação, é possível conectar um gerador de relógio a um contador, com um período de operação conhecido é possível contar incrementos desse período e consequentemente contar a passagem de tempo. Contadores quando empregados desta forma, contando pulsos de PLLs ou circuitos geradores de relógio, são chamados de temporizadores (MAZIDI; CHEN; GHAEMI, 2018). Na Figura 3 temos um contador sendo utilizado como temporizador para contar a passagem de 60 segundos.

Figura 3: Temporizador usado para contar 60 segundos



Fonte: MAZIDI; CHEN; GHAEMI, 2018, p. 147.

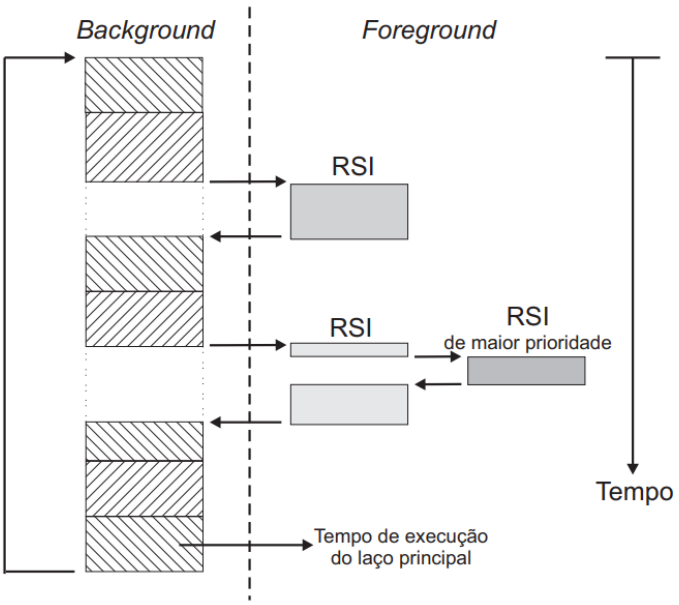
Segundo Wilmshurst (2007, p. 134) “Para uma medição precisa, o início e a parada do contador/temporizador devem ser perfeitamente sincronizados com os eventos. A melhor maneira de fazer isso é usando uma interrupção” (tradução nossa). O autor também afirma que é possível usar temporizadores para medir o tempo entre dois eventos, gerados internamente ou externamente ao microcontrolador. Com isso é possível criar relógios e alarmes periódicos precisos.

2.3.3 Interrupções

Segundo Denardin e Barriquello (2019, p. 49), “Interrupções são a forma de o *hardware* informar que um determinado periférico necessita atenção”. Os autores também afirmam que exemplos comuns são: transição do nível lógico em um pino de entrada ou saída, recepção de dados por uma porta serial ou um temporizador que tem sua contagem expirada. Quando a interrupção ocorre, uma rotina de serviço de interrupção (RSI), projetada pelo desenvolvedor, é executada com o objetivo de atender a demanda gerada pelo periférico a qual a interrupção está ligada.

Utilizando essa ferramenta é possível criar código embarcado primariamente voltado para resposta a ocorrência de eventos, com cada RSI responsável por tratar a ocorrência de um evento, e uma rotina cíclica principal responsável por tarefas menos críticas. Firmware que seguem esse padrão tem a capacidade de aparentar executar duas ou mais tarefas simultaneamente, quando na realidade apenas chaveia entre elas rápido o suficiente (DENARDIN; BARRIQUELLO, 2019). Na Figura 4 temos uma linha do tempo que mostra a execução e chaveamento entre rotinas em código utilizando o recurso de interrupções.

Figura 4: Exemplo de código com interrupções



Fonte: Denardin e Barriquello, 2019.

2.3.4 Protocolo I2C

Este é um protocolo de comunicação amplamente utilizado para conexão de múltiplos periféricos, como sensores, a unidades de processamento. I2C é acrônimo de *Inter-Integrated Circuit* (Inter-Circuito Integrado), e foi desenvolvido pela Phillips Semiconductors, atualmente NXP Semiconductors. É um protocolo utilizado para comunicação de dois ou mais circuitos integrados, como microcontroladores e dispositivos periféricos. Esse padrão de comunicação consiste em um barramento de dois fios, um de dados (SDA) e um de relógio (SCL), que possibilitam transmissão serial bidirecional de 8 bits de dados a até 3.4Mbit/s. (NXP B.V., 2021).

2.3.5 Protocolo SPI

SPI é um protocolo desenvolvido para comunicação entre circuitos integrados em que apenas um dos dispositivos no barramento é o mestre da comunicação, ditando quando e quem deve transferir dados de e para o mestre, que geralmente é um microprocessador embarcado se comunicando com os periféricos, que podem ser sensores ou atuadores. É um protocolo que possui quatro sinais. C*lock* (relógio) também chamado de SCLK, é gerado pelo mestre

da comunicação e dita a velocidade de troca de dados. *Slave select* (selecionador de escravo), também chamado de SS, é responsável por selecionar o dispositivo com quem se deseja comunicar, além disso é o único pino que não faz parte do barramento, cada escravo possui um. *Master Out – Slave In* (Saída mestre - Entrada escravo) também chamado de MOSI, é a linha de dados unidirecional de dados saído do mestre e indo para o escravo. *Master In – Slave Out* (Entrada Mestre – Saída Escravo), também chamado de MISO, é a linha de dados unidirecional de dados saído do escravo e indo para o mestre. Além de concordar em velocidades de relógio, todos os dispositivos do barramento devem atender a mesma configuração de polaridade de *clock* (CPOL), e de fase de amostragem de bit (CPHA) (LEENS, 2009).

2.3.6 Protocolo UART

Este é um dos protocolos disponíveis para comunicação com periféricos em sistemas embarcados, e seu funcionamento foi descrito segundo Wang e Song (2011) da seguinte forma: o formato de transmissão e recepção de dados da UART geralmente inclui o bit de início, os bits de dados, o bit de paridade, o bit de parada e o estado ocioso. O bit de início marca o início da transmissão de dados. Quando o transmissor envia um caractere de dados, um sinal lógico "0" é enviado primeiro, que é o bit de início, e a largura de tempo é um ciclo de relógio de taxa de transmissão. O bit de início é seguido pelos bits de dados, que normalmente variam de 5 a 8 bits. Os bits de dados podem ser seguidos por um bit de paridade, que pode ser paridade ímpar, paridade par ou sem bit de paridade. O bit de paridade ou o bit de dados (quando não há bit de paridade) é seguido pelos bits de parada, que são sinais lógicos "1" contendo 1, 1,5 ou 2 bits. Os bits de parada marcam o fim de um dado.

2.4 PARÂMETROS FÍSICOS, QUÍMICOS E BIOLÓGICOS

Introdução e descrição de parâmetros que são de interesse para a pesquisa.

2.4.1 TVOCs

Do inglês *Total volatile organic compounds* (TVOCs), significando Compostos Orgânicos Voláteis Totais, são uma vasta gama de compostos orgânicos que se encontram na atmosfera. Esses compostos contêm carbono e são emitidos por uma variedade de fontes, tanto naturais como antropogênicas. A definição tradicional diz que esse grupo inclui hidrocarbonetos não metânicos, compostos orgânicos oxigenados e compostos orgânicos halogenados. Eles são cruciais na química atmosférica e afetam a qualidade do ar, desempenhando um papel significativo na formação de ozono e aerossóis orgânicos secundários. A medição dos TVOCs, em vez de apenas monitorar elementos químicos específicos, pode simplificar a modelagem fotoquímica e ajudar no desenvolvimento de estratégias de controle de ozono. Portanto, um monitoramento cuidadoso e contínuo dos TVOCs em ambientes externos é essencial para entender suas fontes, impactos e desenvolver estratégias de mitigação (KOPPMANN, 2007).

2.4.2 TEMPERATURA DO AR

A temperatura é uma medida da energia cinética média das moléculas em uma substância, e é um dos elementos mais fundamentais para entender e monitorar o meio ambiente. Ela influencia diretamente processos físicos, químicos e biológicos na atmosfera, hidrosfera e litosfera. Para uma compreensão mais abrangente, é necessário monitorar a temperatura em diferentes escalas espaciais (microclima, mesoclima e macroclima) e temporais (diária, sazonal e anual). O uso de instrumentos como termômetros, termógrafos, e sensores remotos permite a coleta de dados precisos de temperatura em diversas regiões e altitudes. A partir de uma análise sistemática dos dados de temperatura e outros dados ambientais, é possível realizar projeções climáticas, desenvolver planos de adaptação, e tomar decisões embasadas para um uso mais consciente dos recursos naturais (AHRENS, 2009).

2.4.3 UMIDADE RELATIVA

A umidade, um componente fundamental do sistema climático da Terra, refere-se à quantidade de vapor de água presente na atmosfera. Ela desempenha um papel crucial em diversos processos ambientais, influenciando o ciclo hidrológico, a formação de nuvens e precipitação, o balanço de energia da Terra, estabilidade atmosférica, qualidade do ar, previsão do tempo e conforto humano. Existem várias formas de expressar a umidade, incluindo a pressão de vapor, a umidade relativa, a razão de mistura e a umidade específica. A umidade relativa, em particular, é uma medida comum e expressa a quantidade de vapor de água presente no ar como uma porcentagem da quantidade máxima que o ar pode reter a uma dada temperatura. O monitoramento da umidade é essencial para compreender e prever as mudanças climáticas, os eventos climáticos extremos, e os impactos sobre os ecossistemas e a sociedade (WALLACE, 2006).

2.4.4 PRESSÃO ATMOSFÉRICA

A pressão atmosférica, também conhecida como pressão do ar, é definida como a força exercida pelo peso do ar sobre uma determinada área. Ela influencia o clima, o tempo, a formação de sistemas atmosféricos, a circulação atmosférica, a estagnação do ar e acumulo de poluentes. Além disso é um dos principais parâmetros utilizados em modelos de previsão do tempo. O monitoramento da pressão atmosférica é essencial para a compreensão e a previsão de fenômenos meteorológicos e para a análise das condições climáticas (AHRENS, 2009).

2.5 GERENCIAMENTO DE DISPOSITIVOS E APRESENTAÇÃO DE RESULTADOS

Segundo Oliveira (2017), uma das aplicações mais comuns de Internet das Coisas é funcionar como um registro de dados, ou seja, de apenas registrar uma grandeza obtida. Esse registro pode ter diversas aplicações, como monitoramento ambiental ou aplicações industriais. Como exemplo, um microcontrolador, conectado a um sensor de temperatura periodicamente envia

dados de temperatura para uma API, que são armazenados na nuvem e podem ser visualizados diretamente pela web. Automações baseadas nos valores registrados também são possíveis, ativando alarmes ou acionando atuadores em resposta. Diversos serviços nesse ramo estão disponíveis, com interfaces web customizáveis, contendo gráficos para facilitar análise dos dados coletados, e acesso via internet.

2.5.1 MQTT

Message Queuing Telemetry Transport (MQTT) é um protocolo de comunicação leve e eficiente, amplamente utilizado em aplicações de Internet das Coisas. Ele foi projetado para dispositivos com recursos limitados e redes com alta latência ou largura de banda restrita, tornando-o ideal para cenários de monitoramento e controle remoto (BANKS; GUPTA, 2015).

O protocolo opera sobre o modelo *publish/subscribe*, em que dispositivos chamados "*publishers*" enviam mensagens para tópicos específicos, e "*subscribers*" recebem essas mensagens quando inscritos nos mesmos tópicos. Um componente central do sistema é o broker, responsável por gerenciar o roteamento das mensagens entre publicadores e assinantes. Essa arquitetura desacoplada melhora a escalabilidade e facilita a integração de dispositivos (BANKS; GUPTA, 2015).

3 MÉTODOS E MATERIAIS

Neste capítulo serão apresentados o método utilizado e os componentes selecionados para a construção do sistema. Está é uma pesquisa aplicada de cunho exploratória dos materiais bibliográfico e de laboratório. Foram utilizados procedimentos técnicos de pesquisa e bibliografia experimental. O método de abordagem hipotético-dedutivo e o método de procedimento monográfico foram utilizados em sua elaboração.

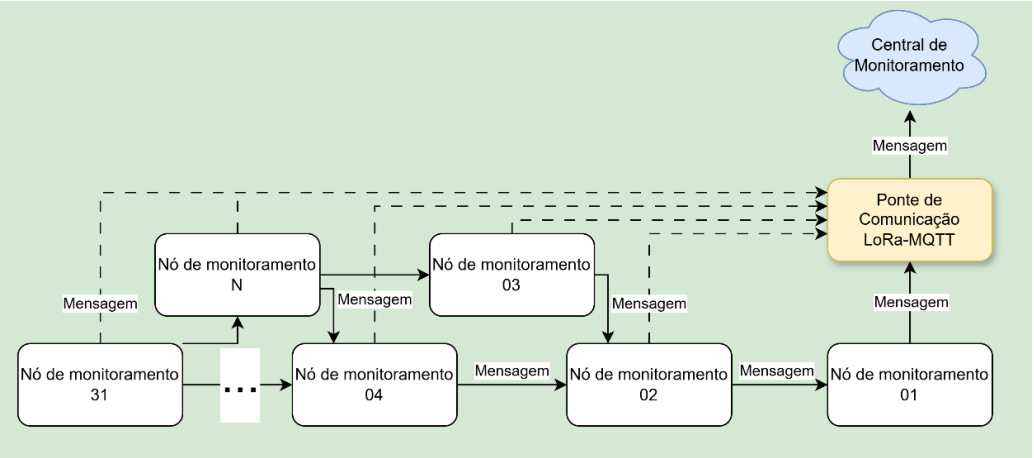
Primeiro é abordada a arquitetura geral do método utilizado para solucionar a problemática na seção 3.1, após isso, na seção 3.2, são descritos os materiais selecionados e suas respectivas funções dentro da solução. E na seção 3.3 é apresentado a utilização dos materiais selecionados nos moldes da arquitetura proposta.

3.1 ARQUITETURA DA SOLUÇÃO

Na Figura 5 podem ser vistos os elementos que compõem a estrutura da rede implementada, assim como a relação entre esses elementos. Nela podemos ver três categorias de elementos da rede: Nós de Monitoramento, Ponte de Comunicação LoRa-MQTT e a Central de Monitoramento.

A estrutura da rede é de uma típica rede em malha, em que cada nó transmite sua própria mensagem de medição, assim como retransmite mensagens de medição recebidas de outros nós, maximizando a possibilidade de um deles conseguir alcançar uma Ponte de Comunicação via radiofrequência LoRa e então ser encaminhada para a Central de Monitoramento via WiFi para a internet.

Figura 5: Arquitetura geral da solução



Fonte: Elaborado pelo autor.

Na mensagem enviada existem metadados que possibilitam evitar a ocorrência de eco infinito na rede, descobrir por quais Nós essa mensagem passou, e qual Nó enviou a mensagem. Uma vez que a mensagem alcança a Ponte de Comunicação, ela é reempacotada para o novo método de transmissão, que é MQTT via radiofrequência WiFi, e então a mensagem finalmente chega na Central Supervisória, onde é salva para posterior apresentação e análise.

Note que apenas existem mensagens direcionadas a Central de Monitoramento, mas não mensagens direcionadas aos Nós. Isso também inclui mensagens de “confirmação de recebimento”, caracterizando a entrega das mensagens enviadas pelos nós como sem garantia, o que não é um problema dada a natureza não crítica das mensagens do cenário do trabalho atual.

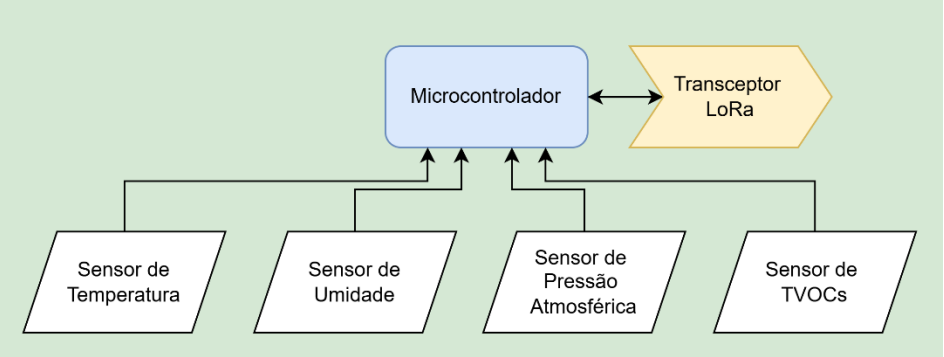
3.1.1 NÓ DE MONITORAMENTO

Este é o dispositivo final instalado no local do monitoramento, responsável por realizar as medições dos sensores a ela conectados, e enviar esses dados para a Ponte de Comunicação.

Na Figura 6 podem ser vistos os principais blocos que compõem esse componente da rede. O microcontrolador é responsável por execução de toda a lógica de operação, e deve ter internamente os periféricos necessários para se

comunicar com cada um dos sensores e transceptor a ele acoplado, conforme necessidade de cada dispositivo.

Figura 6: Arquitetura do Nó de Monitoramento

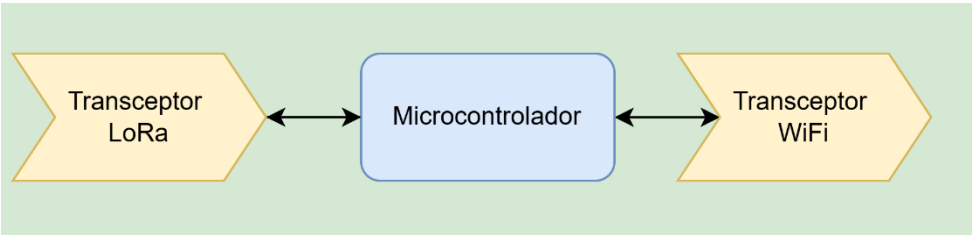


Fonte: Elaborado pelo autor.

3.1.2 PONTE DE COMUNICAÇÃO

Este componente da rede é responsável por receber as mensagens dos Nós de Monitoramento, interpretá-las e as enviar para a Central de Monitoramento. Para isso, além do microcontrolador principal, ele possui dois transceptores, um de LoRa para recebimento das mensagens, e um transceptor de WiFi, para conexão à internet e envio das mensagens para a Central. A arquitetura em blocos da Ponte de Comunicação pode ser vista na Figura *7*.

Figura 7: Arquitetura da Ponte de Comunicação



Fonte: Elaborado pelo autor.

3.2 MATERIAIS

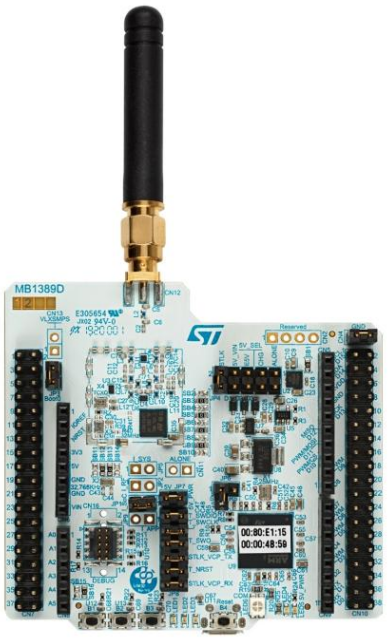
Nesta seção serão apresentados os materiais escolhidos para atender cada uma das necessidades dos componentes do trabalho.

3.2.1 STM32 NUCLEO-WL55JC1

Esta placa de desenvolvimento foi selecionada para ser o microprocessador e transceptor LoRa do Nó de Monitoramento. Além disso, também servirá como transceptor LoRa da ponte de comunicação LoRa-MQTT responsável por encaminhar os dados para o servidor na internet.

A placa, mostrada na Figura 8, é baseada no microprocessador STM32WL55JCI, que possui dois núcleos de processamento, Arm Cortex M4 e M0+ utilizando um *clock* de até 48MHz, possibilitando desenvolvimento de aplicações intrinsecamente paralelas.

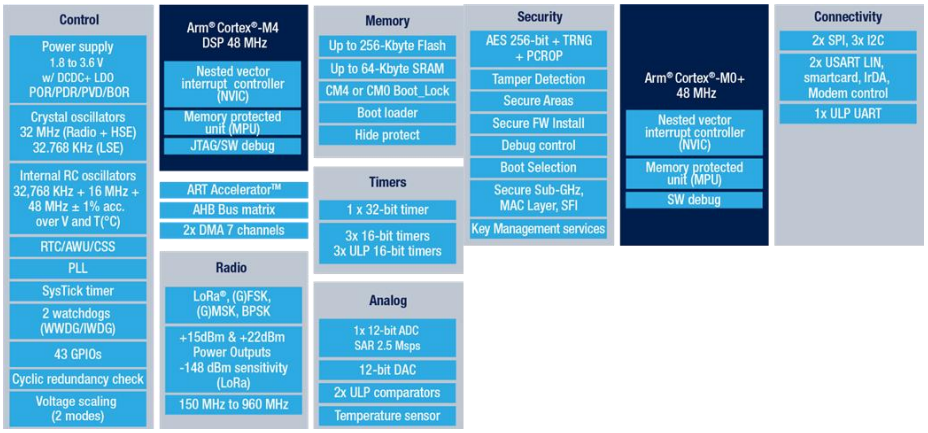
Figura 8: Placa de desenvolvimento NUCLEO-WL55JC1



Fonte: STMicroelectronics, 2024, p. 1.

O microprocessador desta placa conta com os principais periféricos para execução deste trabalho: ADC, temporizador, SPI, I2C, UART, e um transceptor de rádio frequência que dá suporte a modulações LoRa, (G)FSK, (G)MSK, e BPSK, removendo assim a necessidade de um transceptor externo para realizar a comunicação sem fio. A Figura 9 mostra os principais recursos disponíveis nesse microcontrolador.

Figura 9: Principais recursos do microcontrolador STM32WL55JCI



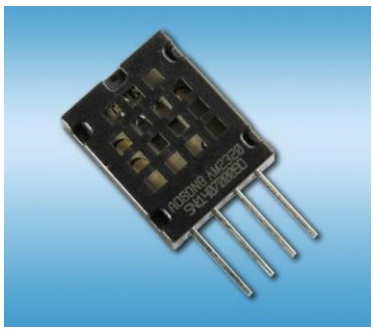
Fonte: Adaptado de STMicroelectronics.

3.2.2 AM2320

Para medição dos parâmetros de temperatura do ar e umidade relativa, foi selecionado o AM2320, que é um sensor digital dessas duas grandezas e pode ser visto na Figura 10. Ele suporta comunicação via I2C e protocolo proprietário sem nome que utiliza um fio. No parâmetro de umidade relativa (UR), possui uma resolução de 0,1%UR, acurácia de ±3%UR, alcance de medição de 0% até 99,9% UR. Já em temperatura possui resolução de 0,1°C, acurácia de ±0,5°C e alcance de -40°C a 80°C (AOSONG ELECTRONICS).

Todas essas características o tornam ideal para a aplicação de monitoramento do ambiente alvo e integração com o microcontrolador escolhido.

Figura 10: AM2320



Fonte: AOSONG ELECTRONICS.

3.2.3 CCS811

O CCS811 foi selecionado como o sensor de medição de compostos orgânicos voláteis totais neste projeto. Mais especificamente foi utilizada a *breakout board*, que pode ser vista na Figura 11, que é uma placa que tem como objetivo disponibilizar os pinos do sensor de modo a facilitar a prototipagem. Este sensor, que se comunica via interface I2C, utiliza a tecnologia *metal oxide semiconductor* (MOX) para detectar gases voláteis no ar, fornecendo leituras de TVOCs em partes por bilhão (ppb) e estimativas de dióxido de carbono equivalente (eCO_2) em partes por milhão (ppm). Sua faixa de medição de TVOCs vai de 0 a 1187 ppb, e de eCO_2 vai de 400 a 8192 ppm.

Figura 11: *Breakout board* do CSS811

Fonte: DIGI-KEY ELECTRONICS.

3.2.4 BMP280

Este é um sensor digital de pressão atmosférica absoluta e temperatura, fabricado pela Bosch Sensortec, e a *breakout board* usada para disponibilizar os pinos do sensor pode ser vista na Figura 12. Sua faixa de medição barométrica vai de 300 hPa até 1100 hPa, correspondente a -500 e +9000 metros em relação ao nível do mar, e de temperatura mede desde -40ºC até +85ºC. Ele apresenta uma precisão de ±1 hPa para pressão e ±0,5°C para temperatura. A comunicação com este sensor pode ser realizada por I2C ou SPI (BOSCH SENSORTEC, 2018).

Essas características permitem sua aplicação no trabalho atual para monitoramento da pressão atmosférica.

Figura 12: *Breakout board* do sensor BMP280

Fonte: Elaborado pelo autor.

3.2.5 ESP32

Para realizar a ponte de comunicação entre rede de radiofrequência LoRa e a rede global de computadores é necessário um segundo microcontrolador, responsável por se comunicar com o primeiro microcontrolador, obter os dados recebidos de outras placas que estão realizando medições a distância, conectar-se à rede global via WiFi e encaminhar esses dados para um *broker* MQTT, onde serão posteriormente tratados.

Para realizar estas atividades, foi selecionado o microcontrolador ESP32, mais especificamente o módulo ESP32-S2-SAOLA-1, que além do microcontrolador, contém outros componentes eletrônicos, como regulador de tensão e circuito de *clock*, entre outros, que são necessários para a operação do microcontrolador. O módulo pode ser visto na Figura 13.

Figura 13: Módulo ESP32-S2-SAOLA-1

Fonte: Espressif Systems, 2025.

3.2.6 STM32CubeIDE

Para realizar o desenvolvimento do código e a programação da placa NUCLEO-WL55JC1, foi utilizado este software, que é um ambiente de desenvolvimento integrado (IDE). Ele permite criação de código para inicialização de periféricos internos ao microcontrolador e controle deles, assim como configuração de *clocks* e definição de pinos. Além disso, conta também com editor de código e compilador e gravador próprio para o microcontrolador sendo utilizado.

3.2.7 ESP IDF

Este é um *framework* de desenvolvimento de código para microcontroladores ESP, desenvolvido e gerenciado pela Espressif Systems e mantido em regime de código aberto. Ele promove uma estrutura base onde o desenvolvedor pode construir suas soluções, utilizando suas ferramentas. Essa estrutura permite controlar os periféricos internos do microcontrolador, e proporciona um ambiente de execução de código em RTOS, um sistema operacional de tempo real, contando criação de *tasks* e obtendo paralelismo funcional. A ferramenta configura, compila e grava o código no microcontrolador, ficando apenas a edição de texto do código a cargo de outra ferramenta.

3.2.8 Autodesk Fusion

Esta é uma plataforma de software baseada na nuvem que disponibiliza ferramentas de CAD, CAM, e design de PCIs. Ela foi utilizada para elaboração do projeto 3D do involucro do Nó de Monitoramento, e exportação de arquivos para fabricação via tecnologia de fabricação aditiva, também conhecida como impressão 3D.

3.2.9 Impressora Creality Ender 3 V2

Para a fabricação das peças que compõem o involucro do Nó de Monitoramento, foi utilizada a impressora 3D Creality Ender 3 V2. Esta é uma impressora do tipo *fused deposition modeling* (FDM), que deposita camadas de material termoplástico para construção de objetos tridimensionais.

O filamento de extrusão utilizado foi o PLA, devido a sua facilidade de impressão, e por ter uma resistência mecânica e uma contração térmica dentro do aceitável para a natureza deste trabalho. A impressora pode ser vista na Figura 14.

Figura 14: Impressora 3D Creality Ender 3 V2

Fonte: Creality, 2025.

3.2.10 THINGSBOARD

Para ser a Central de Monitoramento da rede, foi selecionado o ThingsBoard, que é uma plataforma de código aberto para gerenciamento de dispositivos IoT, coleta e vizualização de dados. Projetada para ser escalável e flexível, ela permite conectar, gerenciar e monitorar dispositivos e sistemas IoT em tempo real. A plataforma oferece suporte a vários protocolos de comunicação, como MQTT, CoAP e HTTP, permitindo a integração com diversos tipos de dispositivos e sensores. Além disso, ele facilita a criação de painéis personalizáveis para visualização dos dados, se tornando uma opção para aplicações de monitoramento remoto, automação industrial e controle ambiental (THINGSBOARD, 2023).

Existem duas maneiras principais de utilização dessa ferramenta: consiste em obter um computador com IP alcançável globalmente e executar o código da ferramenta nesse computador, ou contratar o serviço fornecido pela propria ThingsBoard que executa a ferramenta na nuvem. Após isso apenas é necessário configurar a ferramenta, cadastrar os dispositivos e configurar as telas de monitoramento de cada um deles.

A integração do sistema IoT com esta ferramenta engloba executar a ferramenta, conectar o dispositivo a ela, via MQTT, e trocar mensagens com ela, utilizando o formato JSON especificado em sua documentação de utilização. Além da interface web, a ferramenta tambem disponibiliza APIs para realizar a descarga dos dados de telemetria armazenados na nuvem.

3.3 IMPLEMENTAÇÃO DA SOLUÇÃO

Nessa seção temos as ações tomadas para realizar a interação entre todos os componentes selecionados, conforme especificado na arquitetura da solução.

3.3.1 DEFINIÇÕES DA REDE

A rede implementada se encaixa na definição de redes em malha, entrando assim na categoria de LoRaMESH. Como LoRaMESH não possui órgão responsável por especificar seus detalhes de implementação, os detalhes foram definidos pelo autor.

Os nós de monitoramento da rede transmitem na frequência de 915MHz, com largura de banda de 125KHz, *spreading factor* de 9, *coding rate* de 4/5, e *preamble* de 8 símbolos. As mensagens transmitidas devem conter o cabeçalho mostrado na Tabela 1.

Tabela 1: Cabeçalho do protocolo de mensagens da rede

Item	Tamanho em Bits	Tipo de dado
Tipo da mensagem	8	Inteiro
Identificação do dispositivo	32	Inteiro
Máscara de dispositivos	32	Inteiro

Fonte: Elaborado pelo autor.

O tipo da mensagem indica que tipo de dados estarão disponíveis na mensagem pós cabeçalho. A identificação do dispositivo é um valor único para cada dispositivo, em que apenas 1 bit dos 32 está em valor “1”. A máscara de

dispositivos contém a informação de todos os dispositivos por onde esta mensagem trafegou, com o bit de identificação de cada um deles.

A mensagem de medição pós cabeçalho obedece a Tabela 2.

Tabela 2: Estrutura da mensagem de medição

Item	Tamanho em Bits	Tipo de dado
Tempo ligado	32	Inteiro
Nível da bateria	8	Inteiro
Temperatura	16	Inteiro
Umidade	16	Inteiro
eCO2	16	Inteiro
TVOCs	16	Inteiro
Pressão Atmosférica	16	Ponto Flutuante
Código de alarmes	16	Inteiro
CRC	32	Inteiro

Fonte: Elaborado pelo autor.

Tempo ligado é a quantidade de segundos desde que esse Nó foi ligado. O nível da bateria é um valor que vai de 0 a 100 e é indicativo da tensão atual da bateria. A temperatura, em celsius, e a umidade relativa, em porcentagem, são multiplicadas por 10 antes de serem convertidas para inteiro, assim uma casa decimal é agregada na precisão da medição, mas economizando 32 bits que seriam ocupados caso fosse utilizado ponto flutuante para esses valores. As grandezas de TVOCs, eCO2 e pressão atmosférica são medidas e inalteradas. O código de alarmes contém uma máscara de bits que indicam se houve falha na leitura de algum dos sensores, ou outro problema no sistema. O CRC serve para o receptor confirmar a integridade da mensagem.

3.3.2 NÓ DE MONITORAMENTO

As ligações elétricas do Nó de Monitoramento são mostradas no Apêndice A. Nela são conectadas as linhas de alimentação e terra, o barramento I2C para os sensores CCS811 e BMP280, o sinal de protocolo proprietário do sensor

AM2320, sinais de controle do CCS811, bateria para alimentação do sistema, assim como um divisor resistivo para medição do nível de tensão da bateria.

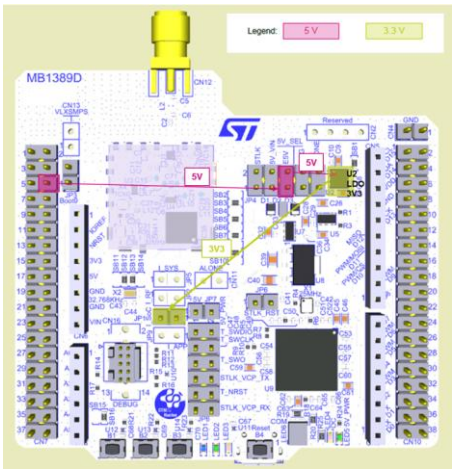
A bateria utilizada foi uma de íon-lítio, com capacidade de 1000mAh. Como característica de baterias desse tipo, sua curva de tensão vai de no máximo 4.2 volts quando carregada, para no mínimo 3 volts quando descarregada. A bateria utilizada pode ser vista na Figura 15.

Figura 15: Bateria de íon-lítio de 1000mAh

Fonte: Elaborado pelo autor.

Para utilizar a bateria em toda a sua curva de descarga, o regulador de tensão já disponível no módulo NUCLEO foi utilizado, a conexão da bateria é feita conforme indicado pelo sinal de “5V” na Figura 16.

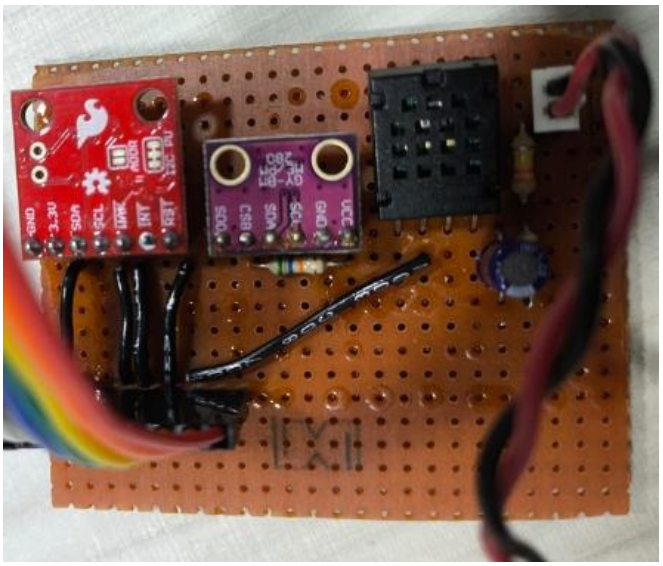
Figura 16: Ponto de conexão da bateria



Fonte: STMicroelectronics, 2024, p. 20.

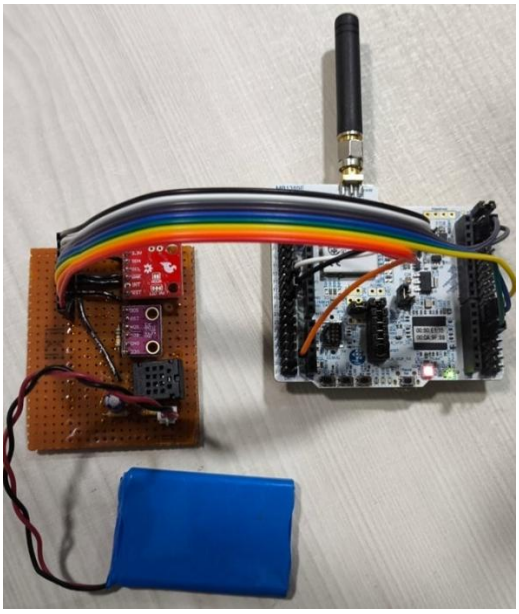
Foi utilizada uma placa universal e fios soldados para realizar as conexões descritas no Apêndice A, a placa universal pode ser vista na Figura 17, e o resultado da conexão na Figura 18.

Figura 17: Placa universal com componentes do Nó de Monitoramento



Fonte: Elaborado pelo autor.

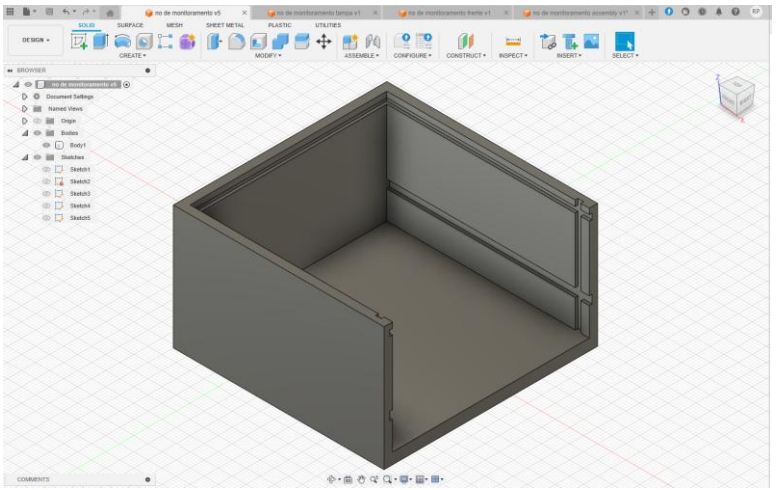
Figura 18: Nó de Monitoramento



Fonte: Elaborado pelo autor.

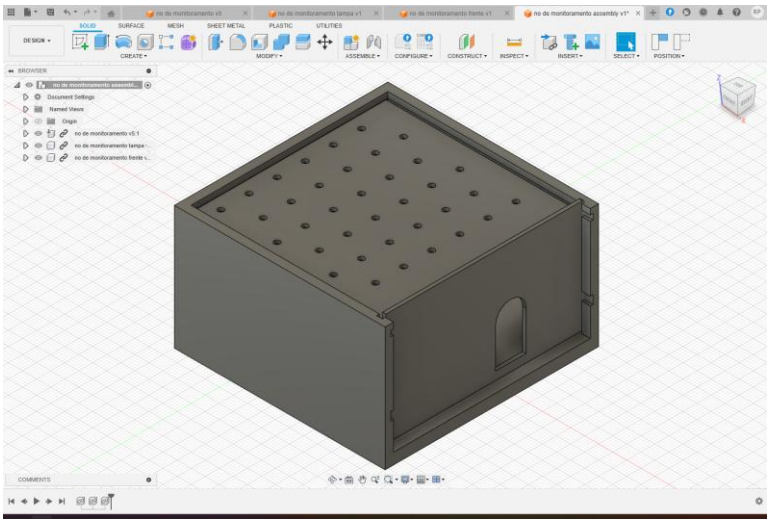
Para ser o involucro físico do Nó de Monitoramento foram elaboradas 3 peças no software Fusion 360. O enclausuramento base pode ser visto na Figura 19, e as três peças, com as duas tampas posicionadas nos seus trilhos, podem ser vistas na Figura 20. A tampa frontal possui furos de respiração para os sensores, e a tampa inferior possui rasgo para a antena do módulo.

Figura 19: Base do invólucro do Nó de Monitoramento



Fonte: Elaborado pelo autor.

Figura 20: Invólucro completo do Nó de Monitoramento

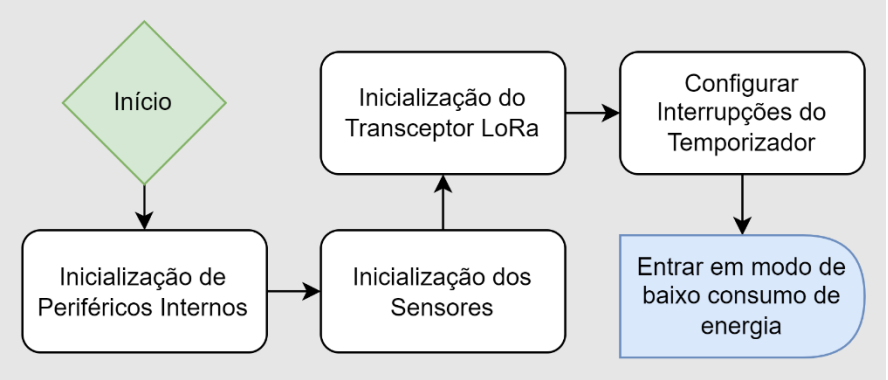


Fonte: Elaborado pelo autor.

A sequência lógica de ações tomadas pelo microprocessador do Nó de Monitoramento logo após ser energizado pode ser vista na Figura 21. Após ser energizado, realiza a inicialização e configuração dos periféricos internos necessários: ADC, I2C, UART e Temporizador. Após isso, é realizada a comunicação com os sensores de temperatura, umidade, pressão e TVOCs, e suas devidas configurações individuais. Em sequência, configuramos o transceptor LoRa com os parâmetros de rede, configurações de interrupção e iniciamos um período de recepção de dados. Com isso pronto, apenas

configuramos o temporizador para gerar uma interrupção no período de amostragem de dados desejado, e após isso entramos em modo de baixo consumo de energia. Como citado na seção 3.2.6, o código de inicialização de periféricos é gerado pelo software, o código de inicialização dos sensores e de configuração do temporizador pode ser visto no Apêndice C. Já o código relacionado ao controle do transceptor LoRa está no Apêndice .

Figura 21: Rotina de inicialização do Nó de Monitoramento

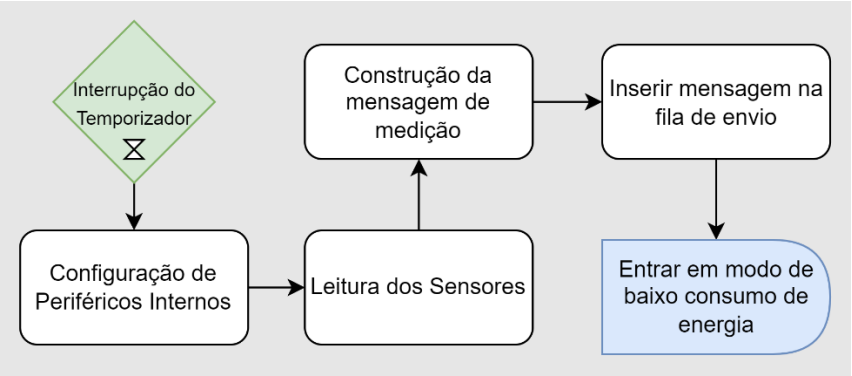


Fonte: Elaborado pelo autor.

Nesse estado, apenas interrupções tem a capacidade de forçar o retorno a execução de instruções do microcontrolador. Abaixo temos as interrupções que foram configuradas no passo de inicialização do sistema, e qual é a sequência lógica de ações após cada uma delas.

Na Figura 22 é mostrada a rotina executada quando o ocorre interrupção por conta do temporizador, que foi configurado o período de amostragem configurável via código. Nessa rotina os periféricos de comunicação que serão utilizados durante a amostragem são reconfigurados, já que o modo de baixo consumo os desabilita. Após isso, é feita a leitura dos sensores conectados, realizando assim a amostragem. Com os valorem em memória, a mensagem de medição é construída e inserida na fila de envio, para envio posterior, e então o dispositivo entra em modo de baixo consumo de energia. Esta rotina esta definida como a função “measurementTask” no Apêndice C.

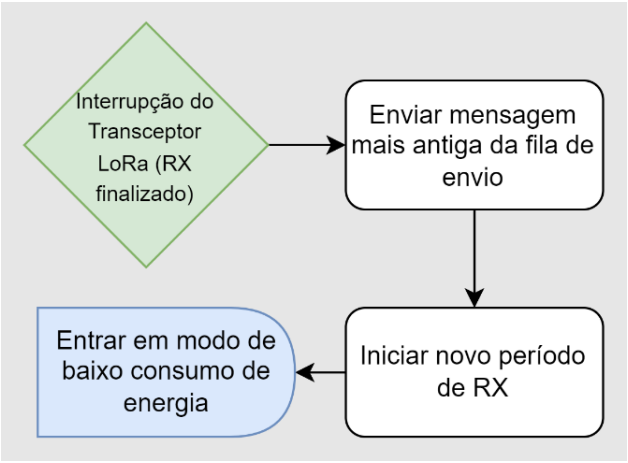
Figura 22: Rotina da interrupção do Temporizador para amostragem de parametros



Fonte: Elaborado pelo autor.

A Figura 23 mostra a rotina executada quando um período de recepção é finalizado, independente de uma mensagem ter sido recebida, ou o período ter finalizado por tempo esgotado. Com o transceptor ocioso, o pacote mais antigo da fila de transmissão é lido e enviado. Após o envio, é iniciado outro período de recepção, seguida pela entrada do microcontrolador em modo de baixo consumo de energia. O código dessa rotina pode ser encontrado na função “radioListenerTask” no Apêndice .

Figura 23: Rotina da interrupção de recepção finalizada

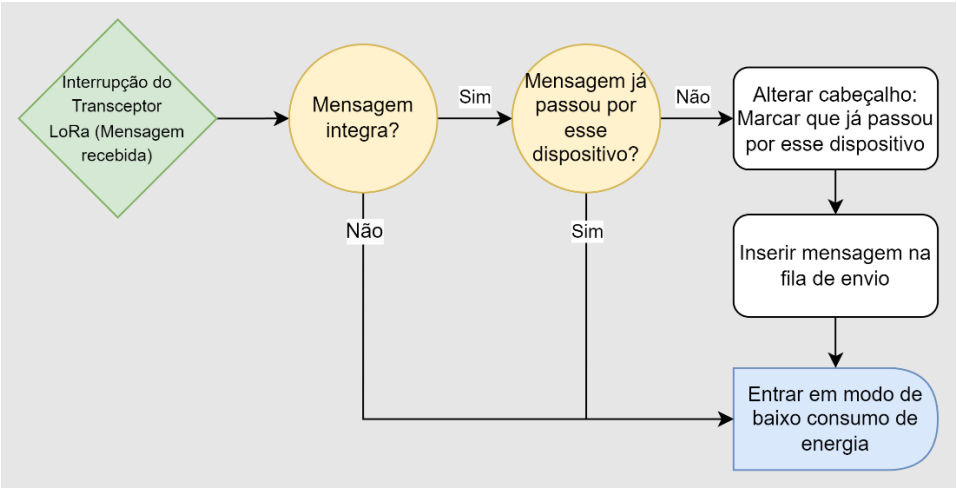


Fonte: Elaborado pelo autor.

Outra rotina de saída do modo de baixo consumo de energia é mostrada na Figura 24, que é executada quando uma mensagem é recebida durante o período de recepção. A integridade da mensagem é checada, e caso esteja

integra é checado no header da própria mensagem se ela já passou por esse dispositivo, caso não tenha passado, o cabeçalho da mensagem é alterado para marcar a passagem por esse Nó, e então a mensagem alterada é inserida na fila de envio. Após isso, ou no caso contrário de qualquer um dos pontos de bifurcação lógica, o dispositivo volta a entrar em modo de baixo consumo de energia. O código dessa rotina consta na função “messageRcvTask” no Apêndice .

Figura 24: Rotina da interrupção de mensagem recebida

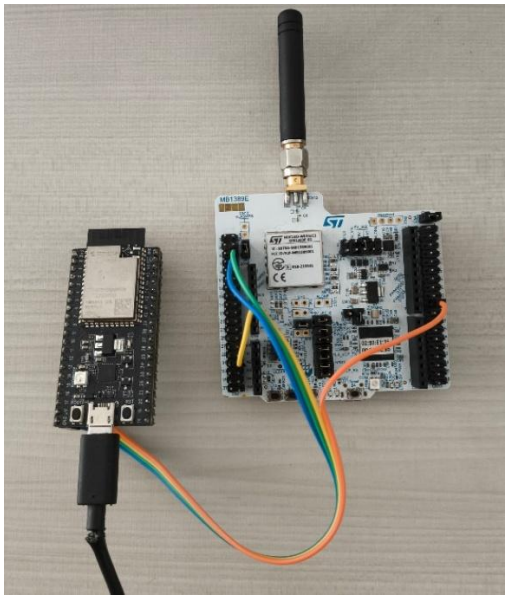


Fonte: Elaborado pelo autor.

3.3.3 PONTE DE COMUNICAÇÃO

As ligações elétricas da Ponte de Comunicação podem ser vistas no Apêndice B. As conexões realizadas são a da UART de comunicação entre os microcontroladores, de alimentação e referência, e do cabo USB que alimentação do sistema. As conexões entre os microcontroladores foram soldadas e o resultado pode ser visto na Figura 25. Para alimentar o sistema, uma fonte genérica de 5 volts com saída USB foi utilizada.

Figura 25: Ponte de Comunicação

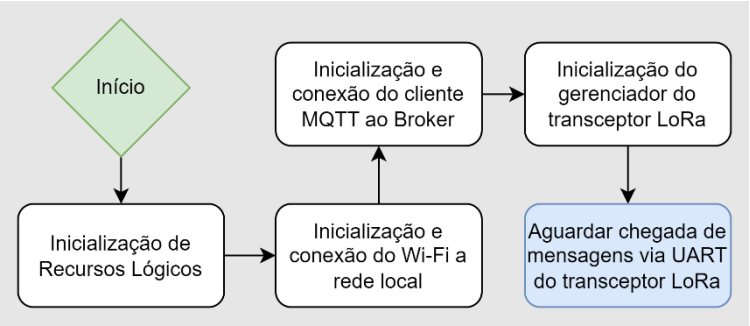


Fonte: Elaborado pelo autor.

Considerando o local alvo de instalação sendo um ambiente interno, ele não precisa de proteções contra os intemperes, e pode operar sem a presença de um invólucro protetor.

A rotina de inicialização do módulo ESP32 da Ponte de Comunicação pode ser vista na Figura 26. Nela, são inicializados recursos necessários para a operação do sistema operacional embarcado, e do Wi-Fi. Após isso, o Wi-Fi é inicializado e configurado para conectar na rede de SSID e senha fixas, definidos via código, e bloqueia a execução até conseguir se conectar. Uma vez conectado, é inicializado e configurado o cliente MQTT, responsável por se conectar ao *broker* da Central de Monitoramento, com a chave de segurança para conexão também gravada no código. É inicializada a UART conectada ao transceptor LoRa, e a comunicação com ele. O código de inicialização está distribuído no Apêndice e no Apêndice .

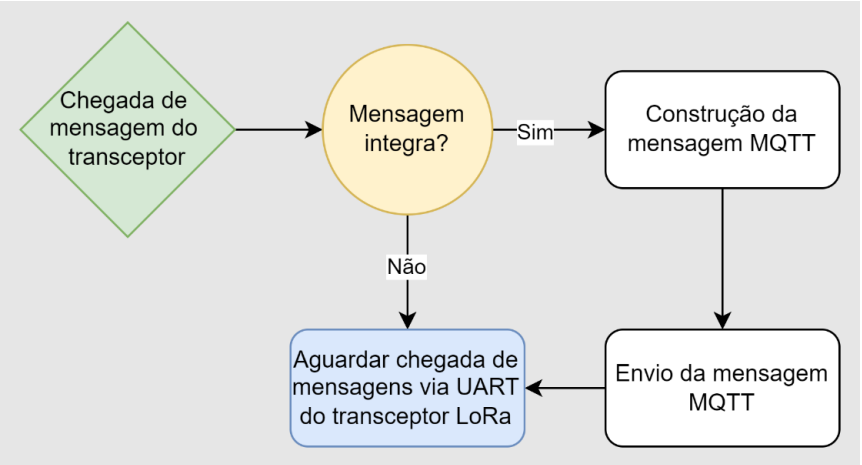
Figura 26: Rotina de inicialização do módulo ESP32 da ponte de comunicação



Fonte: Elaborado pelo autor.

Então, o ESP32 entra em laço perpétuo, aguardando chegada de mensagens dos Nós, e encaminhando essas mensagens para a Central de Monitoramento. Na Figura 27 temos a rotina completa de tratamento de mensagens recebidas do transceptor LoRa. Ao chegar uma mensagem, sua integridade é checada através do CRC presente na mensagem, caso não esteja integra, o processo descarta a mensagem e volta a esperar por outras mensagens. Caso esteja integra, uma mensagem contendo os dados de medição é construída com base na mensagem recebida, porém no formato JSON esperado pela central de monitoramento, e então esta mensagem é enviada para a Central. Após isso, o código volta a esperar pela chegada de outras mensagens via LoRa. O código dessa rotina está nas funções “stmLoraManager” e “processLoraPacket” do Apêndice .

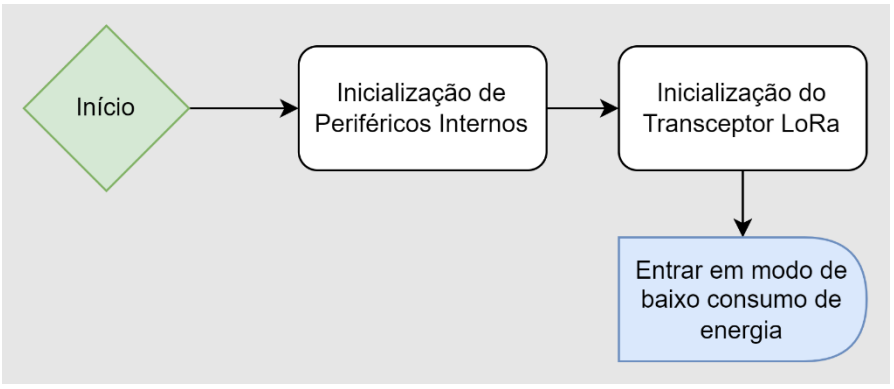
Figura 27: Rotina de tratamento de mensagens recebidas do transceptor LoRa



Fonte: Elaborado pelo autor.

O diagrama em blocos da rotina de inicialização do módulo NUCLEO da ponte de comunicação pode ser visto na Figura 28. Ao ser energizado, o microcontrolado inicializa os periféricos necessários para a operação como Ponte, a UART conectada ao ESP. Após isso, configura o transceptor LoRa interno para recebimento de mensagens e geração de interrupções, e então entra em modo de baixo consumo de energia. O código da rotina está no Apêndice C.

Figura 28: Rotina de inicialização do módulo NUCLEO da ponte de comunicação

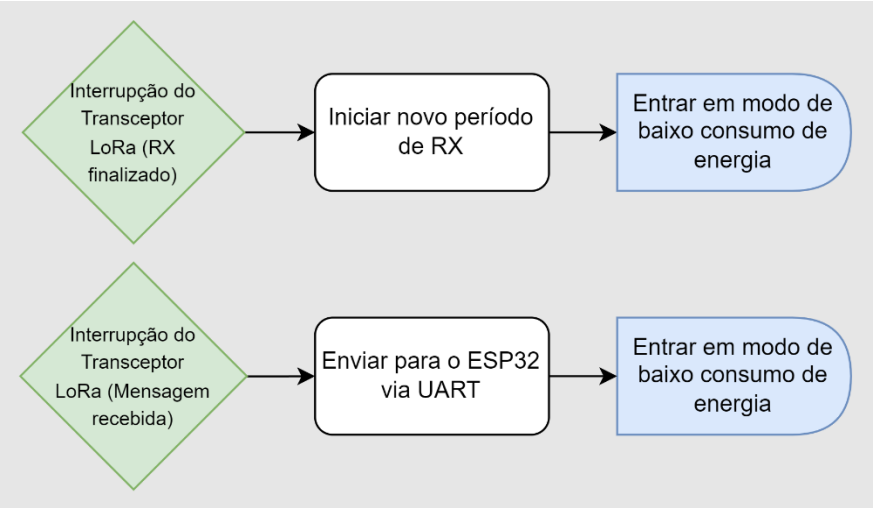


Fonte: Elaborado pelo autor.

Após isso, o NUCLEO apenas executará código quando ocorrer interrupções de tempo esgotado esperando por mensagens, ou de chegada de

mensagens. O diagrama em bloco das duas rotinas pode ser visto na Figura 29. Ambos os códigos das rotinas estão presentes no Apêndice .

Figura 29: Rotinas de interrupções do transceptor LoRa - Ponte de Comunicação



Fonte: Elaborado pelo autor.

3.3.4 CONFIGURAÇÃO DA CENTRAL DE MONITORAMENTO

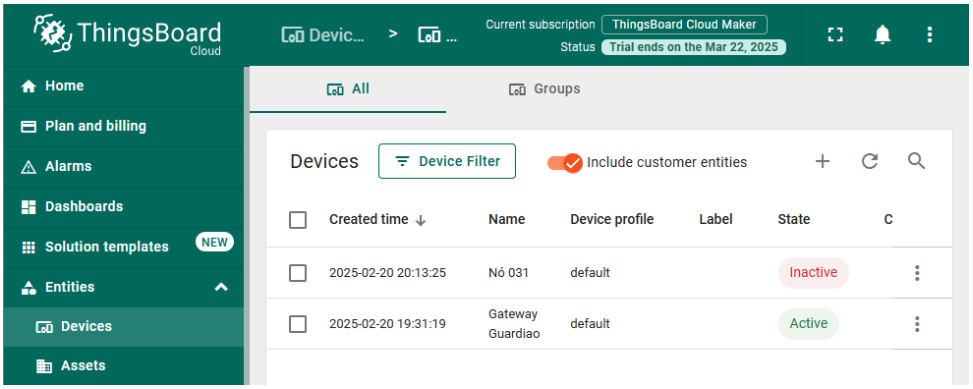
Para realizar a integração do sistema com o serviço ThingsBoard, os seguintes passos foram tomados na interface web do sistema:

1. Criação de conta na ferramenta;
2. Contratação do serviço;
3. Criação do dispositivo Ponte de Comunicação;
4. Criação de uma *Dashboard*;

O passo 1 envolve informar um e-mail e informações pessoais para o contratado, e o passo 2 consiste em realizar o pagamento do plano de utilização do serviço desejado, conforme quantidade de dispositivos.

No passo 3 é onde é gerada a chave de segurança utilizada na conexão MQTT, que é colocada no código fonte da Ponte de Comunicação. Na Figura 30 temos a tela de dispositivos, onde podemos criá-los.

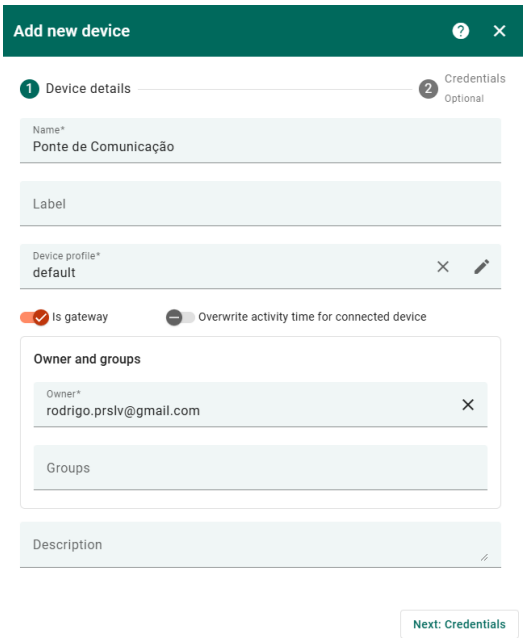
Figura 30: Tela de dispositivos - ThingsBoard



Fonte: Elaborado pelo autor.

Quando criando um dispositivo, a tela mostrada na Figura 31 é exibida. Os pontos importantes no momento da criação da Ponte de Comunicação são o nome do dispositivo, e a opção "*Is gateway*", que marca o dispositivo como um canal por onde outros dispositivos irão enviar mensagens. Os Nós individuais não precisam ser cadastrados, pois por ser cadastrado como um *gateway* a Ponte tem a capacidade de criar os Nós automaticamente no momento que recebe uma mensagem deles.

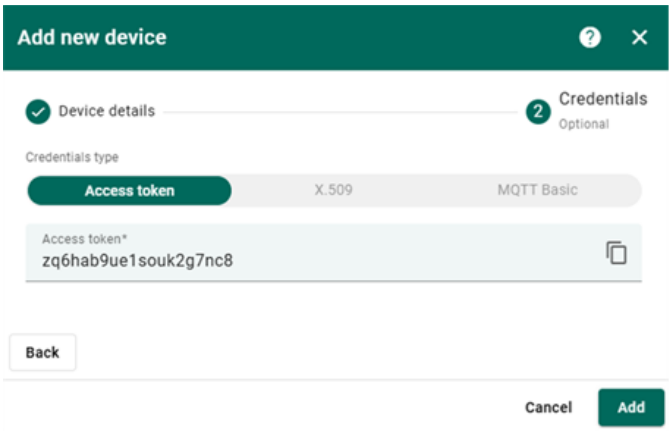
Figura 31: Tela de criação de dispositivos 1 - ThingsBoard



Fonte: Elaborado pelo autor.

Após isso, é exibida uma tela contendo a chave de segurança para conexão ao broker MQTT. Essa tela pode ser vista na Figura 32. Após isso, a adição do dispositivo é finalizada.

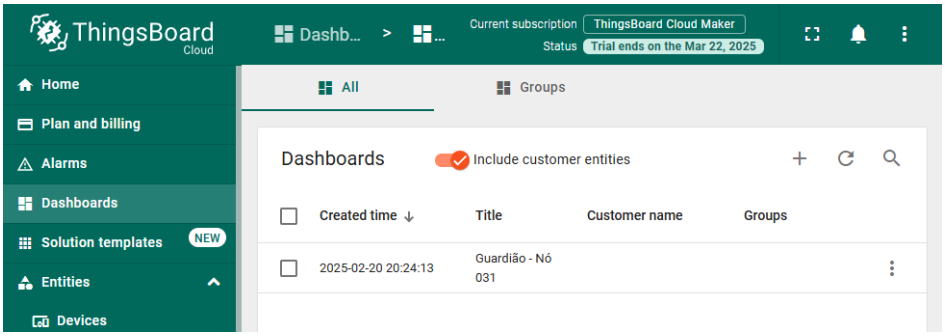
Figura 32: Tela de criação de dispositivos 2 - ThingsBoard



Fonte: Elaborado pelo autor.

O passo 4 envolve posicionar os elementos visuais desejados, para permitir a visualização rápida do estado do sistema e dos últimos valores de monitoramento, dando uma visão geral imediata de um Nó de Monitoramento em específico.

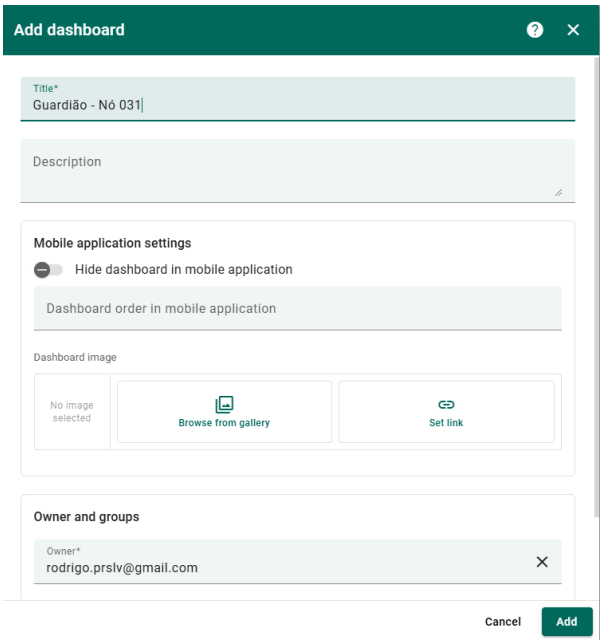
Figura 33: Tela de Dashboards - ThingsBoard



Fonte: Elaborado pelo autor.

A Figura 33 mostra a tela de Dashboards criadas, e permite a criação de novas. Ao iniciar a criação de uma nova, a tela mostrada na Figura 34 é exibida, onde o nome da Dashboard deve ser preenchido. As outras configurações não precisam de alteração.

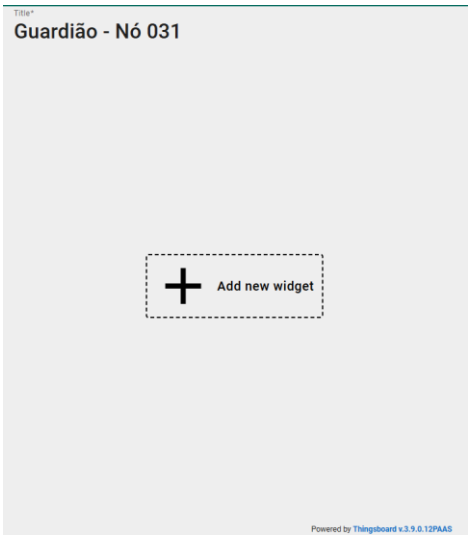
Figura 34: Tela de criação de Dashboard - ThingsBoard



Fonte: Elaborado pelo autor.

Após criação, a dashboard não possui nenhuma informação, e deve ser preenchia com “ferramentas” de monitoramento disponibilizadas pela própria interface web. A Dashboard vazia pode ser vista na Figura 35.

Figura 35: Dashboard vazia - ThingsBoard



Fonte: Elaborado pelo autor.

Após seleção, posicionamento e configuração de dados a serem monitorados nas ferramentas, a Dashboard elaborada pode ser vista na Figura 36. A tela contém todos os últimos valores de qualidade do sinal da mensagem recebida, temperatura, umidade, TVOCs, eCO2, pressão atmosférica, nível de bateria e o código de alarmes. Além dos últimos valores obtidos, também há gráficos com janela de tempo configurável da temperatura, umidade, pressão TVOCs, eCO2 e qualidade de sinal.

Figura 36: Dashboard criada - ThingsBoard

Fonte: Elaborado pelo autor.

3.4 AMBIENTE DE TESTES

Foram realizados testes em dois ambientes: laboratório e campo. Os testes realizados em laboratório foram voltados para validação da rede em malha, e o encaminhamento de mensagens ao longo de múltiplos dispositivos. Os testes em campo visaram validar a transmissão de mensagens em meio a flora densa, típica da Floresta Amazônica.

O teste em laboratório ocorreu com duas unidades de Nó de Monitoramento, e uma unidade de Ponte de Comunicação, posicionados a 5 metros um do outro, em formação triangular, sem barreiras entre eles.

O local do teste em campo é conhecido como Igarapé da Vovó, e fica nas dependências do IFAM CMDI. Na Figura 37 temos uma foto de satélite com as

posições da Ponte de Comunicação, no marcador de 0 metros, e do Nó de monitoramento, no marcador de 105,13 metros.

Figura 37: Posições da Ponte de Comunicação e do Nó de Monitoramento nos testes em campo

Fonte: Google Maps, 2025

A Ponte de Comunicação se encontrava no primeiro andar, dentro da sala de um dos professores, enquanto o Nó de Monitoramento se encontrava fixado a uma das árvores da trilha do Igarapé da Vovó do IFAM CMDI. O Nó fixado pode ser visto na Figura 38.

Figura 38: Nó de Monitoramento fixado em árvore

Fonte: Elaborado pelo autor.

Como a densidade da flora foi essencial para esse teste, a arvore escolhida foi a que contava com a pior linha de visão direta para o local de instalação da Ponte de Comunicação, e que tinha diâmetro suficiente para a fixação do dispositivo. A linha de visão do Nó para a Ponte está ilustrada no Apêndice G, e a visão reversa no Apêndice H.

4 RESULTADOS

Os resultados podem ser separados nos dois cenários de testes especificados na seção 3.4, em laboratório para teste da rede em malha, e em campo, para teste de funcionamento no cenário proposto.

O teste de operação da rede em malha consistiu em posicionar os dispositivos conforme descrito na seção 3.4, e configurar os Nós de Monitoramento para envio de 100 medições via LoRa, com espaçamento entre elas a cada 30 segundos. Os dois Nós foram alimentados simultaneamente. A Ponte de Comunicação estava conectada na Central de Monitoramento e enviando as mensagens recebidas normalmente. Nesse teste as medições dos sensores não são de interesse, apenas a chegada das mensagens íntegras.

Como pode ser visto na Tabela 3, ocorreram 2% e 5% de perda de mensagens nos Nós 01 e 02, respectivamente. A ocorrência dessa perda é atribuída a colisão de pacotes no ar, durante a transmissão, ocasionando a perda dessas mensagens.

Tabela 3: Resultado do teste da rede em malha

Número do Nó	Mensagens Enviadas	Mensagens Recebidas	% de perda
01	100	98	2%
02	100	95	5%

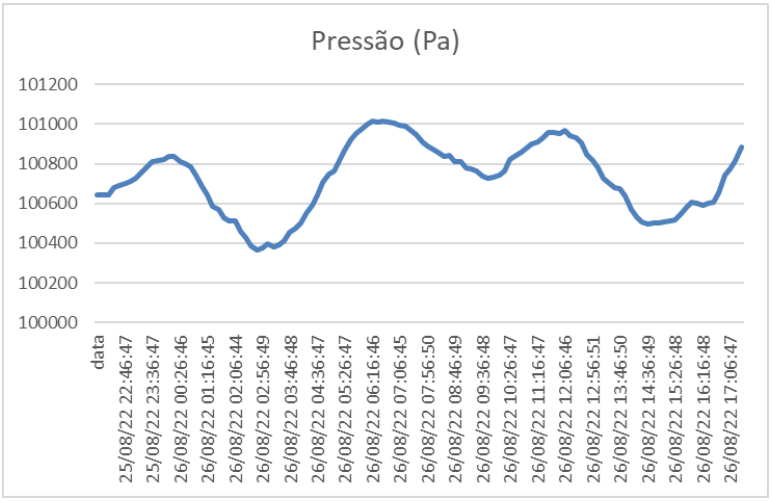
Fonte: Elaborado pelo autor.

Os testes de campo foram iniciados com a chegada da primeira mensagem as 25/08/22 21:56:48 e finalizado as 26/08/22 17:36:50 na última mensagem antes do descarregamento total da bateria, totalizando 19:40:01 de operação. O Nó de Monitoramento estava configurado para envio de medições a cada 10 minutos, e nesse período de teste foram recebidas 118 mensagens que é exatamente o valor esperado, caracterizando uma taxa de perda de mensagens de 0%.

Os gráficos de pressão atmosférica, temperatura, umidade relativa, TVOCs e eCO2 das medições do período do teste podem ser vistos na Figura 39, Figura 40, Figura 41, Figura 42 e Figura 43 respectivamente. Durante o teste

o céu começou nublado, houve leve chuva, e depois o céu ficou limpo sem formação de nuvens.

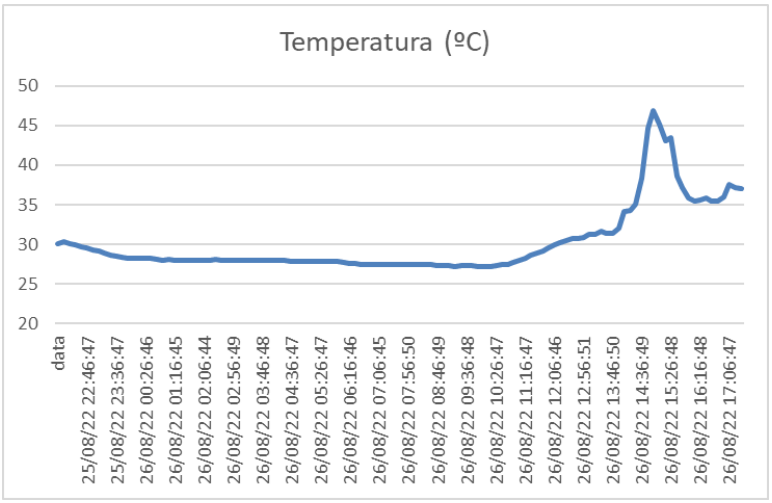
Figura 39: Gráfico de pressão atmosférica



Fonte: Elaborado pelo autor.

Na Figura 40 é possível ver a variação da temperatura conforme a passagem do tempo. Nele, o pico de temperatura ocorreu aproximadamente às 15 horas horário local, como esperado do ambiente de teste.

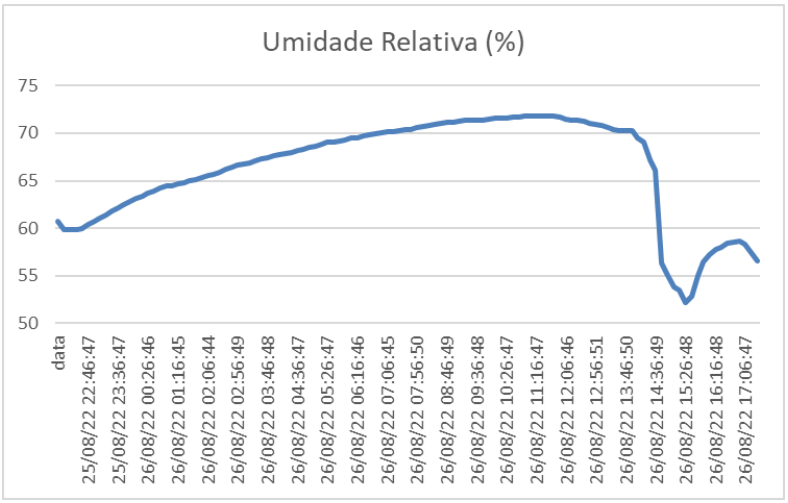
Figura 40: Gráfico de temperatura



Fonte: Elaborado pelo autor.

Na Figura 41 temos a umidade relativa, que caiu conforme o tempo limpou e a incidência solar aumentou.

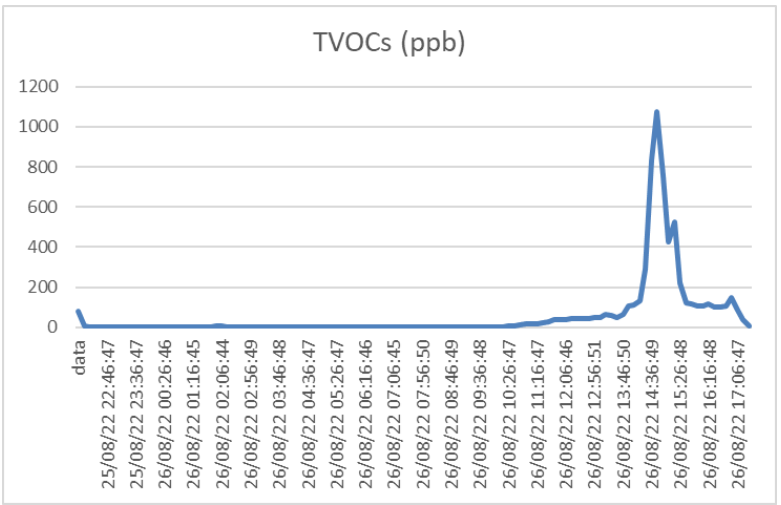
Figura 41: Gráfico de umidade relativa



Fonte: Elaborado pelo autor.

Na Figura 42 podemos observar a concentração de compostos voláteis orgânicos totais aumentar conforme a chuva parou.

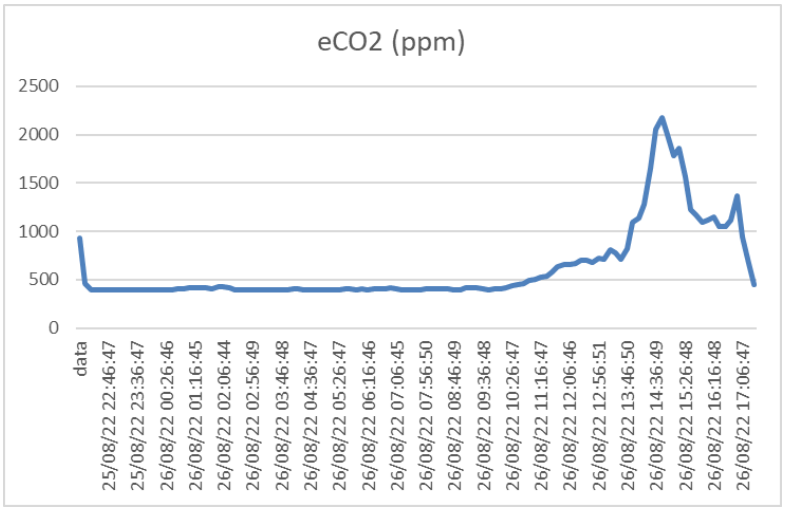
Figura 42: Gráfico de TVOCs



Fonte: Elaborado pelo autor.

E na Figura 43 vemos que o gráfico de CO2 equivalente seguiu a mesma tendência dos compostos orgânicos voláteis totais.

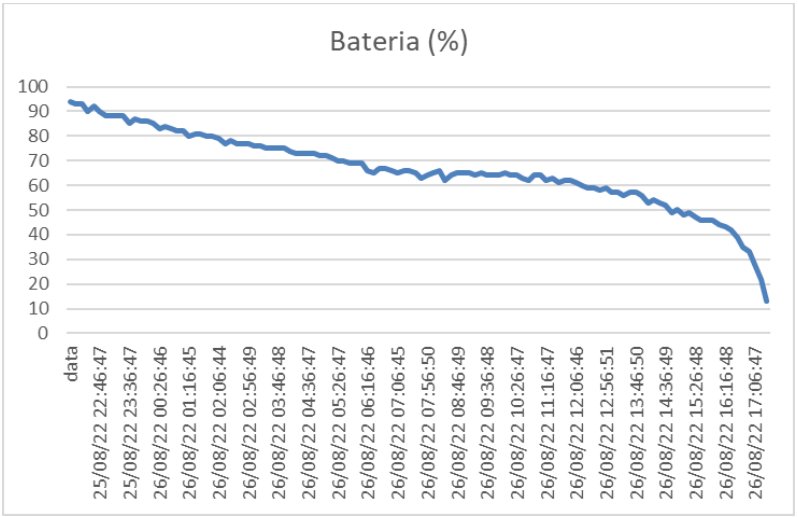
Figura 43: Gráfico de eCO2



Fonte: Elaborado pelo autor.

O gráfico da estimativa de porcentagem de carga de bateria restante durante o teste pode ser visto na Figura 44, nele é notável a curva característica de descarregamento de baterias de íon-lítio.

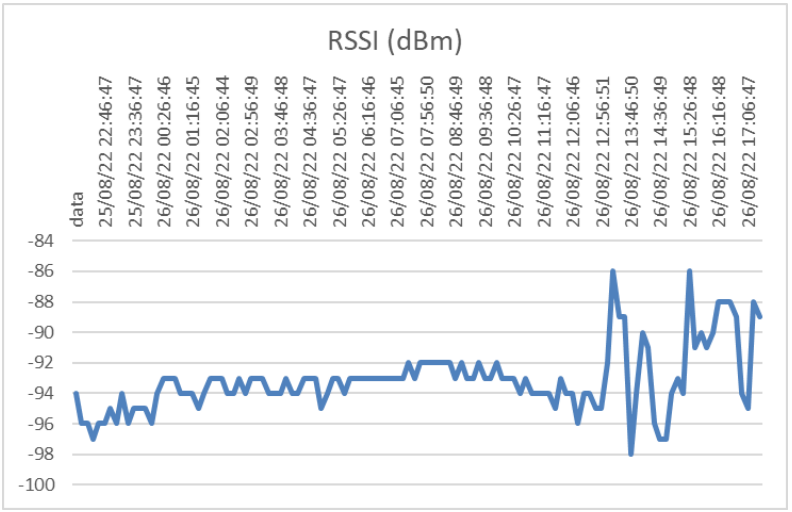
Figura 44: Gráfico de descarga da bateria



Fonte: Elaborado pelo autor.

Na Figura 45 temos o RSSI ao longo do teste, que indica a potência do sinal recebido em cada mensagem. Essa é uma das grandezas afetadas pelo ambiente de teste, denegrindo-a. A média de todas as mensagens ficou em -93,15 dBm, ainda acima da sensibilidade mínima especificada pela ficha técnica do NUCLEO-WL55JC1, que é de -125 dBm.

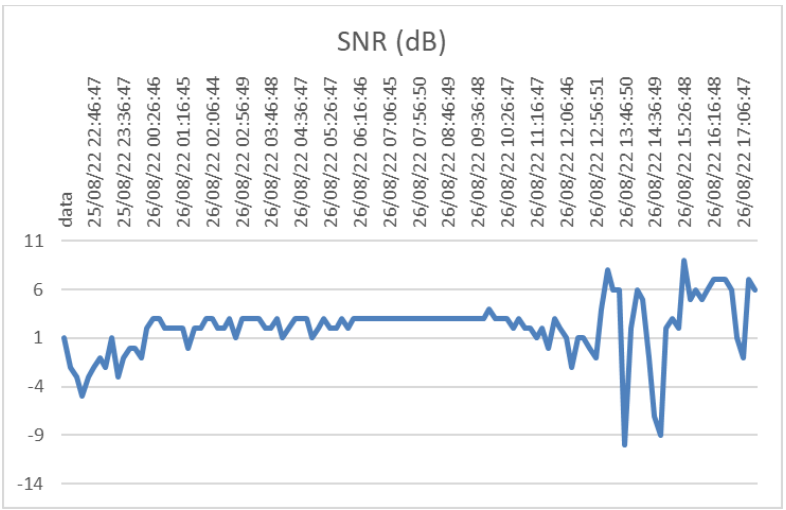
Figura 45: Gráfico de RSSI



Fonte: Elaborado pelo autor.

Na Figura 46 é exibido o nível do *signal to noise ratio* (SNR), que significa a razão entre sinal e ruído, da transmissão de cada mensagem. Com isso, podemos notar que esse cenário também afeta o sinal transmitido nessa métrica, com o mínimo SNR de -9 dB, e uma média de 2,08 dB, próximo do chão de ruído, em 0 dB. Apesar da proximidade ao chão de ruído, a tecnologia LoRa consegue transmitir e receber pacotes abaixo dele, como pode ser visto no gráfico.

Figura 46: Gráfico de SNR



Fonte: Elaborado pelo autor.

Como ponto de comparação ao trabalho atual, existem duas soluções de mercado que realizam uma tarefa semelhante, que serão descritas agora.

O LHT65, que pode ser visto na Figura 47, é um sensor de temperatura e umidade, e permite conexão de outros sensores externos. Conta com uma bateria de 2400mAh e afirma ter uma vida útil de 10 anos. Utiliza LoRaWAN para enviar os dados, deixando a infraestrutura da rede LoRaWAN, como *gateway* LoRaWAN e Central Supervisória, a cargo do comprador (DRAGINO, 2025). As principais diferenças em relação ao trabalho atual são os sensores e a utilização de LoRaWAN, que acarreta a rede não ter estrutura em malha, e, portanto, não pode ser expandida horizontalmente já que depende da distância até o *gateway* para transmitir os dados.

Figura 47: Dragino LHT65 com sensor de luminosidade externo

Fonte: Dragino, 2025.

A segunda solução elvove integrar dois produtos do mesmo fabricante, o WSS-09 com o WSC2-L. O WSS-09, que pode ser visto na Figura 48, contém nove sensores, que são de velocidade e direção do vento, temperatura, umidade, pressão atmosférica, iluminação, MP2.5, MP10, ruído e pluviômetro. Ele foi concebido para ser acoplado ao WSC2-L, que é o transceptor LoRaWAN e pode ser visto na Figura 49. Esse conjunto foi projetado para ser alimentado com uma fonte DC entre 10 e 30 volts (DRAGINO, 2025). Esse conjunto também se difere do trabalho atual por utilizar LoRaWAN. Outro ponto de grande diferença é a

necessidade do sistema precisar ser alimentado constantemente por fonte externa.

Figura 48: Sensor WSS-09

Fonte: Dragino, 2025.

Figura 49: Transmissor WSC2-L acoplado ao sensor WSS-09

Fonte: Dragino, 2025.

5 CONCLUSÕES E TRABALHOS FUTUROS

Utilizando de todo conhecimento apresentado, e através de experimentação, foi possível realizar o desenvolvimento de dispositivo para telemetria e monitoramento remoto de florestas, e alcançar os objetivos propostos pelo trabalho. Este equipamento, caso aprimorado, poderá auxiliar pesquisadores a desenvolverem seus trabalhos de maneira mais fácil e eficiente, e pode vir a se tornar uma nova linha de defesa de florestas.

Como o RSSI ainda ficou acima do aceitável pelo transceptor utilizado, a rede tem a possibilidade de ter os Nós de Monitoramento mais afastados que 100 metros um do outro, expandindo a cobertura máxima da rede.

Durante a realização dos testes em campo, o invólucro do Nó de Monitoramento foi alterado, selando os furos frontais de respiração para evitar entrada de água de chuva. Isso não afetou as leituras por conta do furo da antena na tampa inferior, mas como ponto de melhoria essa alteração deve ser levada em conta, gerando uma nova versão do invólucro, com melhor resistência a chuva. Além disso, o invólucro não conta com método de facilitação de fixação, então a adição de orelhas de fixação beneficiaria o projeto.

Quanto ao gerenciamento energético, ele pode ser aprimorado realizado o design e manufatura de PCI dedicada para esse propósito, com chaveamento da alimentação dos sensores, permitindo maior economia de energia. Outra maneira de estender a vida útil da bateria seria com a inclusão de algum método de colheita de energia, como placa solar ou térmica.

Já na parte lógica da rede em malha proposta, a implementação de listas brancas ou listas negras nos Nós de Monitoramento, através do campo “Identificação do dispositivo” das mensagens, permitiria instalá-los em maior densidade geográfica sem correr o risco de sobrecarregar a rede com retransmissões desnecessárias, e causar perda de pacotes.

REFERÊNCIAS

ALMEIDA, Fernando Rodrigues de. **"VIVALDÃO" X "COHABAM DO PARQUE": Impactos Socioambientais na Cidade de Manaus.** Revista Científica Multidisciplinar Núcleo do Conhecimento. Edição 05. Ano 02, Vol. 01. pp 488-512, Julho de 2017. ISSN:2448-0959.

SARDAR, Muhammad Sohail, et al. Experimental Analysis of LoRa CSS Wireless Transmission Characteristics for Forestry Monitoring and Sensing. **2018 International Symposium in Sensing and Instrumentation in IoT Era (ISSI)**, Shanghai, 2018.

INSTITUTO NACIONAL DE PESQUISAS ESPACIAIS (INPE). obt. inpe, 2020. Deter Intenso. Disponível em: <http://www.obt.inpe.br/OBT/assuntos/programas/amazonia/deter/deter-intenso>. Acesso em: 20/10/2020.

FERREIRA, Ana Elisa, et al. A Visitor Assistance System Based on LoRa for Nature Forest Parks. **Electronics 2020**, 9, 696.

VEGA-RODRÍGUEZ, Roberto, et al. Low Cost LoRa based Network for Forest Fire Detection. **2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)**. Granada, 2019, pp. 177-184.

AHRENS, C. Donald. *Meteorology Today: An Introduction to Weather, Climate, and the Environment*. 9. ed. Australia: Brooks/Cole, Cengage Learning, 2009.

BANKS, Andy; GUPTA, Rahul. *MQTT Version 3.1.1 Plus Errata 01*. OASIS Standard, 2015. Disponível em <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>. Acesso em: 10/06/2023.

OLIVEIRA, Sérgio de. *Internet das Coisas com ESP8266, Arduino e Raspberry Pi.* 1. ed. São Paulo: Novatec Editora, 2017.

WALLACE, John M.; HOBBS, Peter V. *Atmospheric Science: An Introductory Survey*. 2. ed. Amsterdam: Elsevier, 2006.

KOPPMANN, Ralf. *Volatile Organic Compounds in the Atmosphere*. Chichester: John Wiley & Sons, 2007.

FLOYD, Thomas L. *Sistemas digitais: fundamentos e aplicações*. 9. ed. São Paulo: Pearson Prentice Hall, 2006.

ST. **STM32WL Nucleo-64 board (MB1389)**, 2021. Disponível em: <https://www.st.com/resource/en/user_manual/um2592-stm32wl-nucleo64-board-mb1389-stmicroelectronics.pdf>. Acesso em: 25/11/2021.

CENTENARO, Marco, et al. Long-Range Communications in Unlicensed Bands: the Rising Stars in the IoT and Smart City Scenarios. **IEEE Wireless Communications October 2016**. 2016.

VAN EIJK, Patrick. **Internet des objets: zoom sur la technologie réseau sans fil longue portée et basse consommation LoRaWAN**. *L'Embarqué*, 2020. Disponível em: <https://www.lembarque.com/fichiers/cms/file/002_004_APPLICATION_SEMTECH.pdf>. Acesso em: 15 fev. 2025.

PHAM, Congduc; BOUNCEUR, Ahcène; CLAVIER, Laurent; NOREEN, Umber; EHSAN, Muhammad. **Radio channel access challenges in LoRa low-power wide-area networks**. In: CHAUDHARI, Bharat S.; ZENNARO, Marco (Ed.). *LPWAN Technologies for IoT and M2M Applications*. Boca Raton: CRC Press, 2020. p. 65-102.

LORA ALLIANCE. What is LoRaWAN® Specification. **LoRa Alliance**. 2021. Disponível em: <https://lora-alliance.org/about-lorawan/>. Acesso em: 30/11/2021.

SEMTECH. **LoRa™ Modulation Basics (AN1200.22)**, 2015. Disponível em: <https://semtech.my.salesforce.com/sfc/p/#E0000000JelG/a/2R0000001OJu/xvKUc5w9yjG1q5Pb2IIkpolW54YYqGb.frOZ7HQBcRc>. Acesso em 14/02/2025.

MAZIDI, Muhammad Ali; CHEN, Shujen; GHAEMI, Eshragh. **STM32 Arm Programming for Embedded Systems**. 1. ed. MicroDigitalEd, 2018.

WILMSHURST, Tim. **Designing Embedded Systems with PIC Microcontrollers: Principles and Applications**. 1. ed. Amsterdam: Newnes, 2007.

DENARDIN, Gustavo Weber; BARRIQUELLO, Carlos Henrique. **Sistemas Operacionais de Tempo Real e sua Aplicação em Sistemas Embarcados**. 1. ed. São Paulo: Blucher, 2019.

NXP B.V. **UM10204**: I2C-bus specification and user manual. 7.0. rev. 1 Outubro 2021. 62 p. Disponível em: <https://www.nxp.com/webapp/Download?colCode=UM10204&location=null>. Acesso em: 01/12/2022.

BOSCH SENSORTEC. **BMP280: Digital pressure sensor.**: Bosch Sensortec, 2018. Disponível em: <https://www.bosch-sensortec.com/products/environmental-sensors/pressure-sensors/bmp280/>. Acesso em: 01/12/2022.

AOSONG ELECTRONICS. **AM2320: Digital temperature and humidity sensor.**: AOSONG Electronics Co., 2017. Disponível em: <http://www.aosong.com/en/products-41.html>. Acesso em: 02/12/2022.

THINGSBOARD. *IoT platform for data collection, processing and visualization*. Disponível em: <https://thingsboard.io>. Acesso em: 07/06/2023.

CREALITY. *Ender-3 V2 3D Printer*. Disponível em: <https://www.creality.com/products/ender-3-v2-3d-printer-csco>. Acesso em: 24 fev. 2025.

STMICROELECTRONICS. *STM32WL Nucleo-64 board (MB1389)*. UM2592, 2024. Disponível em: <https://www.st.com/resource/en/user_manual/um2592-stm32wl-nucleo64-board-mb1389-stmicroelectronics.pdf>. Acesso em: 24 fev. 2025.

LEENS, Frédéric. An Introduction to I2C and SPI Protocols. **IEEE Instrumentation & Measurement Magazine.** Fevereiro, 2009.

WANG, Yongcheng; SONG, Kefei. A New Approach to Realize UART. **2011 International conference on Electronic & Mechanical Engineering and Information Technology.** Harbin, Agosto de 2011.

DIGI-KEY ELECTRONICS. **SEN-14193 SparkFun Electronics | Development Boards, Kits, Programmers | DigiKey**. Disponível em: <https://www.digikey.com/en/products/detail/sparkfun-electronics/SEN-14193/7066452>. Acesso em 11/01/2025.

ESPRESSIF SYSTEMS. *ESP32-S2-Saola-1 User Guide*. Disponível em: <https://docs.espressif.com/projects/esp-idf/en/v5.1.6/esp32s2/hw-reference/esp32s2/user-guide-saola-1-v1.2.html>. Acesso em: 24 fev. 2025.

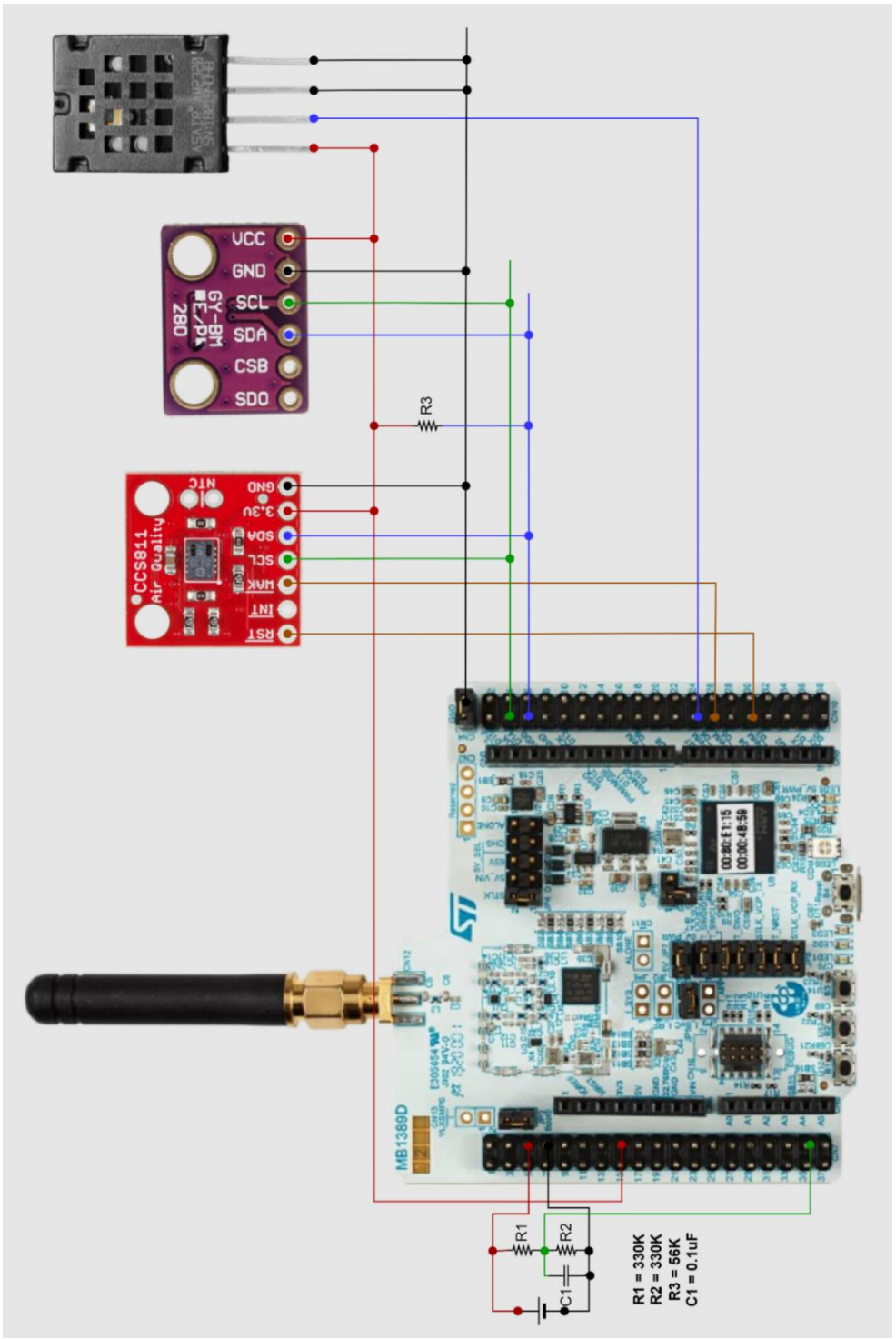
GOOGLE. *Google Maps*. Disponível em: <https://www.google.com/maps/@-3.1351429,-59.9750096,461m/data=!3m1!1e3!5m1!1e4?entry=ttu&g_ep=EgoyMDI1MDIxOS4xIKXMDSoASAFQAw%3D%3D>. Acesso em: 24 fev. 2025.

DRAGINO. LHT65 -- LoRaWAN Temperature & Humidity Sensor. Disponível em: <https://www.dragino.com/products/temperature-humidity-sensor/item/151-lht65.html>. Acesso em: 25 fev. 2025.

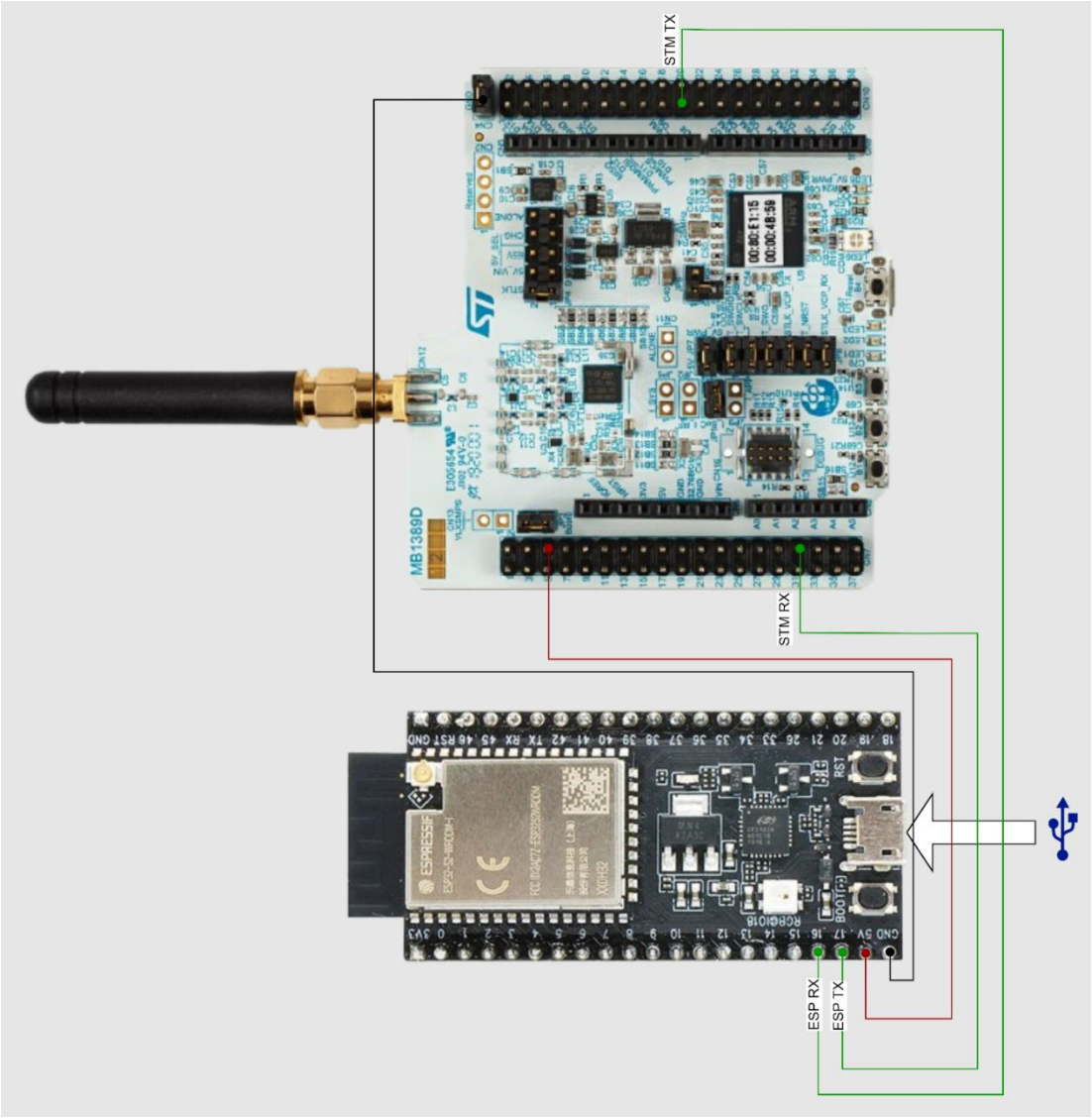
DRAGINO. WSC2-L -- LoRaWAN Weather Station Kit. Disponível em: <https://www.dragino.com/products/agriculture-weather-station/item/339-wsc2-l.html>. Acesso em: 25 fev. 2025.

DRAGINO. WSS-09 -- 9 in 1 Weather Station Sensors. Disponível em: <https://www.dragino.com/products/agriculture-weather-station/item/341-wss-09.html>. Acesso em: 25 fev. 2025.

Apêndice A - Ligação eletrônica do Nó de Monitoramento



Apêndice B - Ligação eletrônica da Ponte de Comunicação



Apêndice C - Código fonte de "g_main.c" – NUCLEO-WL55JC1

```
#include "g_main.h"
#include "g_lora_manager.h"
#include "dht11.h"
#include "ccs811.h"
#include "stm32_ccs811_driver.h"
#include "stm32_rtc.h"
#include "bmp280.h"
#include "crc32.h"

#include <stdint.h>
#include <stdbool.h>
#include <time.h>

// STM INCLUDES
#include "main.h"
#include "tim.h"
#include "sys_app.h"
#include "stm32_timer.h"
#include "stm32_seq.h"
#include "radio.h"
#include "adc.h"

#define G_SET_BIT(REG, BIT_NUM)         ((REG) |= (1UL << BIT_NUM))
#define G_CLEAR_BIT(REG, BIT_NUM)       ((REG) &= ~(1UL << BIT_NUM))
#define G_READ_BIT(REG, BIT_NUM)        ((REG >> BIT_NUM) & 1UL)

#define TIMEOUT_ADC_CONVERSION_MS       200
#define ADC_FULL_SCALE                  (4095.0)
#define ADC_VOLTS_FULL_SCALE            (3.3)
#define ADC_R1                          (330000.0)
#define ADC_R2                          (330000.0)
#define ADC_COMPENSATE_RESISTOR(volts) ((volts) * (ADC_R1 + ADC_R2) / ADC_R2)
#define ADC_TO_VOLTS(adc_raw, vref)    (ADC_COMPENSATE_RESISTOR((adc_raw) * (vref)
/ ADC_FULL_SCALE))
#define BATTERY_MAX_VOLTS               (4.2) /* range de operacao util */
#define BATTERY_MIN_VOLTS               (3.0) /* range de operacao util */
#define MAX_VOLTS_IN_PERCENT            (100.0)
#define MIN_VOLTS_IN_PERCENT            (0.0)
#define ADC_HANDLER                     hadc

/* private variables */

dht11_t dht;
UTIL_TIMER_Object_t measurementTimer;
uint32_t measurementPeriod_ms = 30 * 1000;

float mapf_capped(float in,
                  float in_min, float in_max,
                  float out_min, float out_max)
{
    if (in <= in_min)
        return out_min;
    if (in >= in_max)
        return out_max;
    return (in - in_min) * (out_max - out_min) / (in_max - in_min) + out_min;
}

g_status_t system_get_adc(uint32_t channel, uint16_t *adc_raw_buffer)
{
    HAL_StatusTypeDef err;
    ADC_ChannelConfTypeDef chan =
        {
            .Channel      = channel,
            .Rank         = ADC_REGULAR_RANK_1,
```

```
            .SamplingTime = ADC_SAMPLETIME_160CYCLES_5,
        };

    err = HAL_ADC_Init(&ADC_HANDLER);
    if (err != HAL_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "FAIL HAL_ADC_Init %d\r\n", err);
        return G_FAIL;
    }

    err = HAL_ADC_ConfigChannel(&ADC_HANDLER, &chan);
    if (err != HAL_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "FAIL HAL_ADC_ConfigChannel %d\r\n", err);
        return G_FAIL;
    }

    err = HAL_ADCEx_Calibration_Start(&ADC_HANDLER);
    if (err != HAL_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "FAIL HAL_ADCEx_Calibration_Start %d\r\n", err);
        return G_FAIL;
    }

    err = HAL_ADC_Start(&ADC_HANDLER);
    if (err != HAL_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "FAIL HAL_ADC_Start %d\r\n", err);
        return G_FAIL;
    }

    err = HAL_ADC_PollForConversion(&ADC_HANDLER, TIMEOUT_ADC_CONVERSION_MS);
    if (err != HAL_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "FAIL HAL_ADC_PollForConversion %d\r\n", err);
        return G_FAIL;
    }
    *adc_raw_buffer = HAL_ADC_GetValue(&ADC_HANDLER);

    err = HAL_ADC_Stop(&ADC_HANDLER);
    if (err != HAL_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "FAIL HAL_ADC_Stop %d\r\n", err);
        return G_FAIL;
    }

    err = HAL_ADC_DeInit(&ADC_HANDLER);
    if (err != HAL_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "FAIL HAL_ADC_DeInit %d\r\n", err);
        return G_FAIL;
    }

    return G_OK;
}

g_status_t system_get_adc_average(uint32_t channel, uint16_t *adc_raw_buffer,
unsigned int samples)
{
    uint64_t sum = 0;
    for (int i = 0; i < samples;i++)
    {
        int err = system_get_adc(channel, adc_raw_buffer);
        if (err != G_OK)
        {
            return err;
```

```
        }
        sum += (*adc_raw_buffer);
    }
    sum /= samples;
    *adc_raw_buffer = sum;
    return G_OK;
}

/**
 * @brief
 * @param[out] batt_level em porcentagem (0 - 100)
 *
 */
g_status_t getBatteryLevel(uint8_t *batt_level)
{
    int err;
    uint16_t bat_raw_level;
    err = system_get_adc_average(ADC_CHANNEL_1, &bat_raw_level, 10);
    if (err != G_OK)
    {
        APP_LOG(TS_ON, VLEVEL_L, "FAIL! system_get_adc BAT\r\n");
        return G_FAIL;
    }
    APP_LOG(TS_OFF, VLEVEL_L, "ADC BAT RAW: %d \t", bat_raw_level);

    float vref = ADC_VOLTS_FULL_SCALE;
    uint16_t vref_raw_level;
    if (system_get_adc(ADC_CHANNEL_VREFINT, &vref_raw_level) != G_OK)
    {
        APP_LOG(TS_ON, VLEVEL_L, "FAIL! system_get_adc VREF\r\n");
    }
    else
    {
        vref = __HAL_ADC_CALC_VREFANALOG_VOLTAGE(vref_raw_level,
ADC_RESOLUTION_12B) / 1000.0;
        APP_LOG(TS_OFF, VLEVEL_L, "VREF volts: %f \t", vref);
    }

    float bat_volts = ADC_TO_VOLTS(bat_raw_level, vref);
    APP_LOG(TS_OFF, VLEVEL_L, "BAT volts: %f\r\n", bat_volts);
    *batt_level = (uint8_t)mapf_capped(bat_volts,
                                       BATTERY_MIN_VOLTS, BATTERY_MAX_VOLTS,
                                       MIN_VOLTS_IN_PERCENT, MAX_VOLTS_IN_PERCENT);

    return G_OK;
}

void wakeCcs811()
{
    HAL_GPIO_WritePin(CCS811_WAK_GPIO_Port, CCS811_WAK_Pin, GPIO_PIN_RESET);
    HAL_Delay(20);
}

void sleepCcs811()
{
    HAL_GPIO_WritePin(CCS811_WAK_GPIO_Port, CCS811_WAK_Pin, GPIO_PIN_SET);
}

void enableCcs811()
{
    HAL_GPIO_WritePin(CCS811_RESET_GPIO_Port, CCS811_RESET_Pin, GPIO_PIN_SET);
    wakeCcs811();
}

void disableCcs811()
{
```

```
    sleepCcs811();
    HAL_GPIO_WritePin(CCS811_RESET_GPIO_Port, CCS811_RESET_Pin, GPIO_PIN_RESET);
}

void initBMP280()
{
    if (!BMP280_Check())
    {
        APP_LOG(TS_OFF, VLEVEL_M, "BMP280_Check FAIL\r\n");
    }

    BMP280_Reset();

    // Chip startup time is 2ms (after either power up or soft reset),
    HAL_Delay(3);

    // Version of the chip (BMP280 = 0x58)
    uint8_t ver = BMP280_GetVersion();
    APP_LOG(TS_OFF, VLEVEL_M, "ID=%02X\r\n", ver);

    // Read calibration values
    APP_LOG(TS_OFF, VLEVEL_M, "Calibration: ");
    uint8_t i = BMP280_Read_Calibration();
    APP_LOG(TS_OFF, VLEVEL_M, "[%02X] %s\r\n", i, (i == BMP280_SUCCESS) ? "OK" :
"FAIL");

    // Set normal mode inactive duration (standby time)
    BMP280_SetStandby(BMP280_STBY_1s);

    // Set IIR filter constant
    BMP280_SetFilter(BMP280_FILTER_OFF);

    // Set oversampling for temperature
    BMP280_SetOSRST(BMP280_OSRS_T_x2);

    // Set oversampling for pressure
    BMP280_SetOSRSP(BMP280_OSRS_P_x16);

    BMP280_SetMode(BMP280_MODE_FORCED);
    HAL_Delay(50); // bmp takes 43.2 ms to read

    // Chip working mode
    i = BMP280_GetMode();
    APP_LOG(TS_OFF, VLEVEL_M, "Mode: [%02X] -> ", i);
    switch (i)
    {
        case BMP280_MODE_SLEEP:
            APP_LOG(TS_OFF, VLEVEL_M, "SLEEP\r\n");
            break;
        case BMP280_MODE_FORCED:
        case BMP280_MODE_FORCED2:
            APP_LOG(TS_OFF, VLEVEL_M, "FORCED\r\n");
            break;
        default:
            APP_LOG(TS_OFF, VLEVEL_M, "NORMAL\r\n");
            break;
    }
}

g_status_t getTempAndHumid(uint16_t *temp, uint16_t *humid)
{
    int tries = 3;
    while (tries-- > 0)
    {
        uint8_t err = readDHT11(&dht);
        if (err == 1)
```

```
        {
            // APP_LOG(TS_OFF, VLEVEL_M, "temp: %d, humid: %d)\r\n",
dht.temperature, dht.humidity);
            memcpy(temp, &dht.temperature, sizeof(dht.temperature));
            memcpy(humid, &dht.humidity, sizeof(dht.humidity));
            return G_OK;
        }
        else
        {
            APP_LOG(TS_ON, VLEVEL_M, "Falha lendo dht11 (tries: %d)\r\n", tries);
            HAL_Delay(1000);
        }
    }
    return G_FAIL;
}

g_status_t getAirQuality(uint16_t *eCO2, uint16_t *TVOCs)
{
    int tries = 3;

    wakeCcs811();
    HAL_Delay(1100);

    while (tries-- > 0 && ccs.available() == false)
    {
        APP_LOG(TS_ON, VLEVEL_M, "CCS sem dados novos. (tries: %d)\r\n", tries);
        HAL_Delay(2000);
    }

    if (ccs.available() == true)
    {
        tries = 3;
        while (tries-- > 0)
        {
            uint8_t err = ccs.readData();
            if (err == 0)
            {
                *eCO2  = ccs.geteCO2();
                *TVOCs = ccs.getTVOC();
                sleepCcs811();
                return G_OK;
            }
            else
            {
                APP_LOG(TS_ON, VLEVEL_M, "CCS: erro lendo dados! (err=%d)\r\n",
err);
            }
        }
        APP_LOG(TS_ON, VLEVEL_L, "CCS: erro lendo dados!\r\n");
    }

    sleepCcs811();
    return G_FAIL;
}

g_status_t getPressure(float *pressure)
{
    uint8_t i;
    int32_t UT, UP;
    float temperature_f;
    float pressure_f;
    *pressure = 0.0;

    BMP280_SetMode(BMP280_MODE_FORCED);
    HAL_Delay(50); // bmp takes 43.2 ms to read
```

```
    // Check status of chip
    i = BMP280_GetStatus();
    APP_LOG(TS_ON, VLEVEL_ALWAYS, "Status: [%02X] %s %s\r\n",
            i,
            (i & BMP280_STATUS_MEASURING) ? "MEASURING" : "READY",
            (i & BMP280_STATUS_IM_UPDATE) ? "NVM_UPDATE" : "NVM_READY");

    // Get raw readings from the chip
    i = BMP280_Read_UTP(&UT, &UP);
    if (UT == 0x80000)
    {
        // Either temperature measurement is configured as 'skip' or first
conversion is not completed yet
        APP_LOG(TS_OFF, VLEVEL_ALWAYS, "Temperature: no data\r\n");
        // There is no sense to calculate pressure without temperature readings
        APP_LOG(TS_OFF, VLEVEL_ALWAYS, "Pressure: no temperature readings\r\n");
        return G_FAIL;
    }

    // Temperature (must be calculated first)
    temperature_f = BMP280_CalcTf(UT);
    APP_LOG(TS_OFF, VLEVEL_ALWAYS, "BMP Temperature: %i.%dC\r\n",
(int32_t)temperature_f, (uint32_t)((temperature_f - (int32_t)temperature_f) *
100.0));

    // Pressure
    if (UP == 0x80000)
    {
        // Either pressure measurement is configured as 'skip' or first conversion
is not completed yet
        APP_LOG(TS_OFF, VLEVEL_ALWAYS, "Pressure: no data\r\n");
        return G_FAIL;
    }
    pressure_f = BMP280_CalcPf(UP);
    APP_LOG(TS_OFF, VLEVEL_ALWAYS, "BMP Pressure: %u.%uPa\r\n",
(uint32_t)pressure_f, (uint32_t)((pressure_f - (uint32_t)pressure_f) * 1000.0));

    *pressure = pressure_f;
    return G_OK;
}

void measurementTask()
{
    static measurement_message_t measurementMessage;
    reInitClocks();
    memset(&measurementMessage, 0, sizeof(measurement_message_t));
    measurementMessage.original_device_id = DEVICE_ID;
    measurementMessage.devices_mask       = DEVICE_ID;
    measurementMessage.message_type       = MEASUREMENT_MESSAGE;
    APP_LOG(TS_OFF, VLEVEL_M, "\r\n");
    APP_LOG(TS_ON, VLEVEL_M, " >>> measurementTask <<< \r\n");
    APP_LOG(TS_OFF, VLEVEL_H, "DEVICE_ID: 0x%08X\r\n",
measurementMessage.original_device_id);

    if (getEpoch(&measurementMessage.epoch) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "measurementTask: getEpoch FAIL!\r\n");
        G_SET_BIT(measurementMessage.alarms, ALARM_FAIL_READING_RTC);
    }
    APP_LOG(TS_OFF, VLEVEL_H, "epoch: %d \r\n", measurementMessage.epoch);

    if (getBatteryLevel(&measurementMessage.batteryLevel) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "measurementTask: getBatteryLevel FAIL!\r\n");
        G_SET_BIT(measurementMessage.alarms, ALARM_FAIL_READING_BATTERY);
    }
```

```
    APP_LOG(TS_OFF, VLEVEL_H, "batteryLevel: %d \r\n",
measurementMessage.batteryLevel);

    if (getTempAndHumid(&measurementMessage.temperature,
&measurementMessage.humidity) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "measurementTask: getTempAndHumid FAIL!\r\n");
        G_SET_BIT(measurementMessage.alarms, ALARM_FAIL_READING_DHT);
    }
    APP_LOG(TS_OFF, VLEVEL_H, "temperature: %d \r\nhumidity: %d \r\n",
            measurementMessage.temperature,
            measurementMessage.humidity);

    if (getAirQuality(&measurementMessage.eCO2, &measurementMessage.TVOCs) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "measurementTask: getAirQuality FAIL!\r\n");
        G_SET_BIT(measurementMessage.alarms, ALARM_FAIL_READING_CCS811);
    }
    APP_LOG(TS_OFF, VLEVEL_H, "eCO2: %d \r\nTVOCs: %d \r\n",
            measurementMessage.eCO2,
            measurementMessage.TVOCs);

    if (getPressure(&measurementMessage.pressure) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "measurementTask: getPressure FAIL!\r\n");
        G_SET_BIT(measurementMessage.alarms, ALARM_FAIL_READING_BMP);
    }
    APP_LOG(TS_OFF, VLEVEL_ALWAYS, "Pressure: %fPa\r\n",
measurementMessage.pressure);

    APP_LOG(TS_OFF, VLEVEL_H, "alarms: %d \r\n", measurementMessage.alarms);

    measurementMessage.crc32 = xcrc32((uint8_t *)&measurementMessage,
sizeof(measurementMessage) - sizeof(measurementMessage.crc32),
DEFAULT_CRC_INIT_SEED);
    APP_LOG(TS_OFF, VLEVEL_H, "crc: 0x%08X\r\n", measurementMessage.crc32);

    APP_LOG(TS_OFF, VLEVEL_H, "\r\n");
    uint8_t *p = (uint8_t *)&measurementMessage;
    for (int i = 0; i < sizeof(measurement_message_t); i++)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "%02X", p[i]);
        if (i % 16 == 15)
        {
            APP_LOG(TS_OFF, VLEVEL_H, "\r\n");
        }
    }
    APP_LOG(TS_OFF, VLEVEL_H, "\r\n");

    if (sendLoraMessage(&measurementMessage) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "Falha enfileirando pacote para ser
enviado!\r\n");
    }
}

static void onMeasurementTimerEvent(void *context)
{
    UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_Measurement), CFG_SEQ_Prio_0);
    UTIL_TIMER_Start(&measurementTimer);
}

void read_all()
{
    static measurement_message_t measurementMessage;
    memset(&measurementMessage, 0, sizeof(measurement_message_t));
```

```
    if (getEpoch(&measurementMessage.epoch) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "read_all_loop: getEpoch FAIL!\r\n");
    }
    APP_LOG(TS_OFF, VLEVEL_H, "epoch: %d \r\n", measurementMessage.epoch);

    if (getBatteryLevel(&measurementMessage.batteryLevel) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "read_all_loop: getBatteryLevel FAIL!\r\n");
    }
    APP_LOG(TS_OFF, VLEVEL_H, "batteryLevel: %d \r\n",
measurementMessage.batteryLevel);

    if (getTempAndHumid(&measurementMessage.temperature,
&measurementMessage.humidity) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "read_all_loop: getTempAndHumid FAIL!\r\n");
    }
    APP_LOG(TS_OFF, VLEVEL_H, "temperature: %d \t humidity: %d \r\n",
            measurementMessage.temperature,
            measurementMessage.humidity);

    if (getAirQuality(&measurementMessage.eCO2, &measurementMessage.TVOCs) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "read_all_loop: getAirQuality FAIL!\r\n");
    }
    APP_LOG(TS_OFF, VLEVEL_H, "eCO2: %d \t TVOCs: %d \r\n",
            measurementMessage.eCO2,
            measurementMessage.TVOCs);

    if (getPressure(&measurementMessage.pressure) != G_OK)
    {
        APP_LOG(TS_OFF, VLEVEL_L, "read_all_loop: getPressure FAIL!\r\n");
    }
    APP_LOG(TS_OFF, VLEVEL_ALWAYS, "Pressure: %fPa\r\n",
measurementMessage.pressure);
    APP_LOG(TS_OFF, VLEVEL_ALWAYS, "\r\n\r\n");
}

g_status_t guardiaoInit()
{
    APP_LOG(TS_OFF, VLEVEL_M, "\r\n\r\n >>> guardiao_init <<<\r\n");

    // init sensores
    init_dht11(&dht, &htim16, DHT_GPIO_Port, DHT_Pin);

    enableCcs811();
    ccs.begin(CCS811_ADDRESS, &stm32_css811_driver);
    ccs.setDriveMode(CCS811_DRIVE_MODE_1SEC);

    initBMP280();

    // init tasks
    UTIL_SEQ_RegTask((1 << CFG_SEQ_Task_Measurement), UTIL_SEQ_RFU,
measurementTask);

    // init timers
    UTIL_TIMER_Create(&measurementTimer, measurementPeriod_ms, UTIL_TIMER_ONESHOT,
onMeasurementTimerEvent, NULL);

#if defined(GATEWAY_MODE) && (GATEWAY_MODE == 1)
    // todo: coisas pra fazer apenas no modo gateway

#elif defined(GATEWAY_MODE) && (GATEWAY_MODE == 0) /* modo dispositivo final */
    // coloca o timer da measurement task pra rodar
```

```
    UTIL_TIMER_Start(&measurementTimer);
#endif

    // init comunicacao
    loraManagerInit();

    // debug:
    UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_Measurement), CFG_SEQ_Prio_0);
    // measurementTask();

    return G_OK;
}
```

Apêndice D - Código fonte de "g_lora_manager.c" – NUCLEO-WL55JC1

```
#include "g_lora_manager.h"
#include "g_main.h"
#include "crc32.h"
#include "esp_gtw.h"

#include "main.h"
#include "sys_app.h"
#include "radio.h"
#include "stm32_seq.h"
#include "stm32_timer.h"

// AU915
#define RF_FREQUENCY 915000000 /* Hz */

#define TX_OUTPUT_POWER 14      /* dBm */
#define LORA_BANDWIDTH 0        /* [0: 125 kHz, 1: 250 kHz, 2: 500 kHz, 3:
Reserved] */
#define LORA_SPREADING_FACTOR 9 /* [SF7..SF12] */
#define LORA_CODINGRATE 1       /* [1: 4/5, 2: 4/6, 3: 4/7, 4: 4/8] */
#define LORA_PREAMBLE_LENGTH 8  /* Same for Tx and Rx */
#define LORA_SYMBOL_TIMEOUT 5   /* Symbols */
#define LORA_FIX_LENGTH_PAYLOAD_ON false
#define LORA_IQ_INVERSION_ON false
#define TX_MESSAGE_LIST_SIZE 5

#define RX_TIMEOUT_VALUE 5000
#define TX_TIMEOUT_VALUE (int)(RX_TIMEOUT_VALUE * 1.5)
#define MAX_APP_BUFFER_SIZE 255

// pra facilitar telemetria de qualidade do sinal
#define INVALID_RSSI 42 // so de ser > 0 ja eh invalido
#define INVALID_SNR 42  // so de ser > 14 ja eh invalido

typedef struct
{
    uint8_t isEmpty;
    measurement_message_t message;
} msg_controller_t;

static RadioEvents_t RadioEvents;
static uint8_t BufferRx[MAX_APP_BUFFER_SIZE];
```

```
static uint16_t RxBufferSize = 0;
static int16_t RssiValue = 0;
static int8_t SnrValue = 0;
static int32_t random_delay;
static msg_controller_t txMsgList[TX_MESSAGE_LIST_SIZE];

/* gerenciamento de txMsgList */

void initMsgController()
{
    for (int i = 0; i < TX_MESSAGE_LIST_SIZE; i++)
    {
        txMsgList[i].isEmpty = true;
    }
}

msg_controller_t *getOldestMessageController()
{
    msg_controller_t *oldest = &txMsgList[0];
    for (int i = 0; i < TX_MESSAGE_LIST_SIZE; i++)
    {
        if (txMsgList[i].isEmpty == false)
        {
            if (oldest->isEmpty == true || txMsgList[i].message.epoch < oldest-
>message.epoch)
            {
                oldest = &txMsgList[i];
            }
        }
    }

    if (oldest->isEmpty == true)
    {
        return NULL;
    }
    return oldest;
}

void removeOldestMessage()
{
    msg_controller_t *oldest = getOldestMessageController();
    if (oldest == NULL)
    {
        return;
```

```
    }
    oldest->isEmpty = true;
}

measurement_message_t *getOldestUnsentMessage()
{
    msg_controller_t *oldest = getOldestMessageController();
    if (oldest == NULL)
    {
        return NULL;
    }
    return &oldest->message;
}

g_status_t insertMsgToSend(measurement_message_t *message)
{
    for (int i = 0; i < TX_MESSAGE_LIST_SIZE; i++)
    {
        if (txMsgList[i].isEmpty == true)
        {
            txMsgList[i].isEmpty = false;
            memcpy((uint8_t *)&txMsgList[i].message, message,
sizeof(measurement_message_t));
            return G_OK;
        }
    }
    return G_FAIL;
}

// gerenciamento de pacotes tx e rx

void sendOldestUnsentMessage(void)
{
    measurement_message_t *message;
    message = getOldestUnsentMessage();
    if (message != NULL)
    {
        APP_LOG(TS_ON, VLEVEL_M, "ENVIANDO PACOTE DE EPOCH %d\r\n", message-
>epoch);
        Radio.Send((uint8_t *)message, sizeof(measurement_message_t));
    }
}

void messageRcvTask(void)
```

```
{
    APP_LOG(TS_OFF, VLEVEL_H, "Rssi=%d dBm, Snr=%ddB\n\r", RssiValue, SnrValue);
    APP_LOG(TS_OFF, VLEVEL_H, "RxBufferSize=%d \n\r", RxBufferSize);
    measurement_message_t *rxMeasure = (measurement_message_t *)BufferRx;
    uint32_t calculatedCRC = xcrc32(BufferRx, RxBufferSize - sizeof(uint32_t),
DEFAULT_CRC_INIT_SEED);
    uint32_t receivedCRC = rxMeasure->crc32;

    for (int i = 0; i < RxBufferSize; i++)
    {
        APP_LOG(TS_OFF, VLEVEL_H, "%02X", BufferRx[i]);
        if (i % 16 == 15)
        {
            APP_LOG(TS_OFF, VLEVEL_H, "\n\r");
        }
    }
    APP_LOG(TS_OFF, VLEVEL_H, "\r\n");

#if defined(GATEWAY_MODE) && (GATEWAY_MODE == 1)
    gtw_measurement_message_t gtw_msg = {
        .rssi = RssiValue,
        .snr = SnrValue,
        .lowestRssi = INVALID_RSSI,
        .lowestSnr = INVALID_SNR,
        .lowestRssiSender = INVALID_DEVICE_ID,
        .lowestSnrSender = INVALID_DEVICE_ID,
    };
    memcpy(&gtw_msg, BufferRx, sizeof(measurement_message_t));
    gtw_msg.crc32 = xcrc32((uint8_t *)&gtw_msg, sizeof(gtw_measurement_message_t) -
sizeof(uint32_t), DEFAULT_CRC_INIT_SEED);
    reInitClocks();
    sendToEsp((uint8_t *)&gtw_msg, sizeof(gtw_measurement_message_t));
#endif

    if (calculatedCRC == receivedCRC)
    {
        APP_LOG(TS_OFF, VLEVEL_M, " ---CRC VALIDADO---!\r\nMensagem:\r\n\r\n");
        APP_LOG(TS_OFF, VLEVEL_H, "DEVICE_ID: 0x%08X\r\n", rxMeasure-
>devices_mask);
        APP_LOG(TS_OFF, VLEVEL_H, "epoch: %d \r\n", rxMeasure->epoch);
        APP_LOG(TS_OFF, VLEVEL_H, "batteryLevel: %d \r\n", rxMeasure-
>batteryLevel);
        APP_LOG(TS_OFF, VLEVEL_H, "temperature: %d \r\nhumidity: %d \r\n",
                rxMeasure->temperature,
```

```
                rxMeasure->humidity);
        APP_LOG(TS_OFF, VLEVEL_H, "eCO2: %d \r\nTVOCs: %d \r\n",
                rxMeasure->eCO2,
                rxMeasure->TVOCs);
        APP_LOG(TS_OFF, VLEVEL_H, "pressure: %d \r\nalarms: %d \r\n",
                rxMeasure->pressure,
                rxMeasure->alarms);

#if defined(GATEWAY_MODE) && (GATEWAY_MODE == 0) /* modo dispositivo final */
        // transmite a mensagem se essa ela nunca foi transmitida por esse
dispositivo
        if ((rxMeasure->devices_mask & DEVICE_ID) == 0)
        {
            rxMeasure->devices_mask |= DEVICE_ID;
            sendLoraMessage(rxMeasure);
        }
#endif
    }
    APP_LOG(TS_OFF, VLEVEL_M, "receivedCRC: 0x%08X; calculatedCRC: 0x%08X;\r\n",
receivedCRC, calculatedCRC);
    memset(BufferRx, 0, MAX_APP_BUFFER_SIZE);
}

static void OnTxDone(void)
{
    APP_LOG(TS_ON, VLEVEL_L, "OnTxDone\n\r");
    removeOldestMessage();
}

static void OnRxDone(uint8_t *payload, uint16_t size, int16_t rssi, int8_t
LoraSnr_FskCfo)
{
    APP_LOG(TS_OFF, VLEVEL_M, "\r\n");
    APP_LOG(TS_ON, VLEVEL_M, " >>> MENSAGEM RECEBIDA <<<\n\r");
    RssiValue = rssi;
    SnrValue = LoraSnr_FskCfo;
    RxBufferSize = size;

    if (RxBufferSize <= MAX_APP_BUFFER_SIZE)
    {
        memcpy(BufferRx, payload, RxBufferSize);
        UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_MessageRcv), CFG_SEQ_Prio_0);
    }
    else
```

```
    {
        APP_LOG(TS_OFF, VLEVEL_H, "Payload recebido maior que o limite\r\n");
    }

    UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_RadioListener), CFG_SEQ_Prio_1);
}

static void OnTxTimeout(void)
{
    APP_LOG(TS_ON, VLEVEL_L, "OnTxTimeout ================== FATAL
==================\n\r");
}

static void OnRxTimeout(void)
{
    APP_LOG(TS_ON, VLEVEL_L, "OnRxTimeout\n\r");
    UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_RadioListener), CFG_SEQ_Prio_0);
}

static void OnRxError(void)
{
    APP_LOG(TS_ON, VLEVEL_L, "OnRxError\n\r");
    UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_RadioListener), CFG_SEQ_Prio_0);
}

void radioListenerTask()
{
    APP_LOG(TS_ON, VLEVEL_H, " >>> radioListenerTask <<<\r\n");
    sendOldestUnsentMessage();
    /* radio buga se der rx logo em seguida e nao faz o tx */
    HAL_Delay(1000);
    Radio.Rx(RX_TIMEOUT_VALUE + random_delay);
}

g_status_t sendLoraMessage(measurement_message_t *message)
{
    return insertMsgToSend(message);
}

g_status_t loraManagerInit()
{
    /* Radio initialization */
    RadioEvents.TxDone = OnTxDone;
    RadioEvents.RxDone = OnRxDone;
```

```
    RadioEvents.TxTimeout = OnTxTimeout;
    RadioEvents.RxTimeout = OnRxTimeout;
    RadioEvents.RxError = OnRxError;

    Radio.Init(&RadioEvents);

    /*calculate random delay for synchronization*/
    random_delay = (Radio.Random()) >> 22; /*10bits random e.g. from 0 to 1023 ms*/

    Radio.SetChannel(RF_FREQUENCY);

    /* Radio configuration */
    APP_LOG(TS_OFF, VLEVEL_M, "---------------\n\r");
    APP_LOG(TS_OFF, VLEVEL_M, "LORA_MODULATION\n\r");
    APP_LOG(TS_OFF, VLEVEL_M, "LORA_BW=%d kHz\n\r", (1 << LORA_BANDWIDTH) * 125);
    APP_LOG(TS_OFF, VLEVEL_M, "LORA_SF=%d\n\r", LORA_SPREADING_FACTOR);

    Radio.SetTxConfig(MODEM_LORA, TX_OUTPUT_POWER, 0, LORA_BANDWIDTH,
                      LORA_SPREADING_FACTOR, LORA_CODINGRATE,
                      LORA_PREAMBLE_LENGTH, LORA_FIX_LENGTH_PAYLOAD_ON,
                      true, 0, 0, LORA_IQ_INVERSION_ON, TX_TIMEOUT_VALUE);

    Radio.SetRxConfig(MODEM_LORA, LORA_BANDWIDTH, LORA_SPREADING_FACTOR,
                      LORA_CODINGRATE, 0, LORA_PREAMBLE_LENGTH,
                      LORA_SYMBOL_TIMEOUT, LORA_FIX_LENGTH_PAYLOAD_ON,
                      0, true, 0, 0, LORA_IQ_INVERSION_ON, true);

    Radio.SetMaxPayloadLength(MODEM_LORA, MAX_APP_BUFFER_SIZE);

    initMsgController();

    //
    UTIL_SEQ_RegTask((1 << CFG_SEQ_Task_MessageRcv), UTIL_SEQ_RFU, messageRcvTask);
    UTIL_SEQ_RegTask((1 << CFG_SEQ_Task_RadioListener), UTIL_SEQ_RFU,
radioListenerTask);
    UTIL_SEQ_SetTask((1 << CFG_SEQ_Task_RadioListener), CFG_SEQ_Prio_0);

    return G_OK;
}
```

Apêndice E – Código fonte de “main.c” – ESP32

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "sdkconfig.h"
#include "esp_err.h"
#include "esp_netif.h"
// #define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE
#include "esp_log.h"

#include "crc32.h"
#include "stm32wl.h"
#include "mqtt_manager.h"

#define SHINITAI                1
#define LORA_MANAGER_STACK_SIZE 4096
#define LORA_MANAGER_PRIORITY   5
#define MAX_LORA_PKG_SIZE       1024

static const char *TAG = "main";
static uint8_t packageReceived[1024];
static char packageToSend[1024];

int find_set_bit(uint32_t value)
{
    if (value == 0 || (value & (value - 1)) != 0)
    {
        return -1; // No bits set or multiple bits set
    }

    int position = 0;
    while (value > 1)
    {
        value >>= 1;
        position++;
    }
    return position;
}

void processLoraPacket(uint8_t *pkg, uint16_t size)
{
    ESP_LOGI(TAG, ">>> processLoraPacket <<<");
    if (size != sizeof(gtw_measurement_message_t))
    {
        ESP_LOGE(TAG, "size diferente! rxSize: %d; expectedSize: %d", size,
sizeof(gtw_measurement_message_t));
        return;
    }

    gtw_measurement_message_t *rxMeasure;
    rxMeasure = (gtw_measurement_message_t *)pkg;

    int calculatedCrc = xcrc32(pkg, size - sizeof(rxMeasure->crc32),
DEFAULT_CRC_INIT_SEED);
    if (rxMeasure->crc32 != calculatedCrc)
    {
        ESP_LOGW(TAG, "LORA-GTW crc diferente! rxCrc32:%d; calculatedCrc: %d",
rxMeasure->crc32, calculatedCrc);
    }

    int device_number    = find_set_bit(rxMeasure-
>measurement_message.original_device_id);
    char device_name[64] = {0};
    snprintf(device_name, sizeof(device_name) - 1, "Nó %03d", device_number);
    snprintf(packageToSend, sizeof(packageToSend) - 1,
```

```
"{\"%s\":[{\
\"DEVICES_MASK\": %d, \
\"epoch\": %d,\
\"batteryLevel\": %d, \
\"temperature\": %f, \
\"humidity\": %f, \
\"eCO2\": %d, \
\"TVOCs\": %d, \
\"pressure\": %f, \
\"alarms\": %d, \
\"rssi\": %d, \
\"snr\": %d\
}]}",
            device_name,
            rxMeasure->measurement_message.devices_mask,
            rxMeasure->measurement_message.epoch,
            rxMeasure->measurement_message.batteryLevel,
            rxMeasure->measurement_message.temperature / 10.0,
            rxMeasure->measurement_message.humidity / 10.0,
            rxMeasure->measurement_message.eCO2,
            rxMeasure->measurement_message.TVOCs,
            rxMeasure->measurement_message.pressure,
            rxMeasure->measurement_message.alarms,
            rxMeasure->rssi,
            rxMeasure->snr);

    calculatedCrc = xcrc32((uint8_t *)&rxMeasure->measurement_message,
sizeof(measurement_message_t) - sizeof(rxMeasure->crc32), DEFAULT_CRC_INIT_SEED);
    if (rxMeasure->measurement_message.crc32 != calculatedCrc)
    {
        ESP_LOGW(TAG, "END-DEVICE crc diferente! rxCrc32:%d; calculatedCrc: %d",
rxMeasure->crc32, calculatedCrc);
    }
    else
    {
        mqtt_send(packageToSend);
    }
    ESP_LOGI(TAG, "%s", packageToSend);
}

void stmLoraManager(void *params)
{
    ESP_LOGI(TAG, ">>> stmLoraManager <<<");
    stm32UartInit();

    int bytesRead = 0;
    while (1)
    {
        waitForPackage(packageReceived, MAX_LORA_PKG_SIZE, &bytesRead);
        processLoraPacket(packageReceived, bytesRead);
    }
}

void app_main(void)
{
    mqtt_init();
    xTaskCreate(stmLoraManager, "stmLoraManager", LORA_MANAGER_STACK_SIZE, NULL,
LORA_MANAGER_PRIORITY, NULL);
}
```

Apêndice F - Código fonte de "mqtt_manager.c" - ESP32

```
#include <stdio.h>
#include <stdint.h>
#include <stddef.h>
#include <string.h>
#include "esp_wifi.h"
#include "esp_system.h"
#include "nvs_flash.h"
#include "esp_event.h"
#include "esp_netif.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "freertos/queue.h"
#include "lwip/sockets.h"
#include "lwip/dns.h"
#include "lwip/netdb.h"
// #define LOG_LOCAL_LEVEL ESP_LOG_VERBOSE
#include "esp_log.h"
#include "mqtt_client.h"

#include "mqtt_manager.h"
#include "wifi_manager.h"

static const char *TAG = "mqtt_manager";
esp_mqtt_client_handle_t mqtt_client;
int connected = false;

static void log_error_if_nonzero(const char *message, int error_code)
{
    if (error_code != 0)
    {
        ESP_LOGE(TAG, "Last error %s: 0x%x", message, error_code);
    }
}

static esp_err_t mqtt_event_handler_cb(esp_mqtt_event_handle_t event)
{
    esp_mqtt_client_handle_t client = event->client;
    int msg_id;
    switch (event->event_id)
    {
    case MQTT_EVENT_CONNECTED:
        ESP_LOGI(TAG, "MQTT_EVENT_CONNECTED");
        connected = true;
        break;
    case MQTT_EVENT_DISCONNECTED:
        ESP_LOGI(TAG, "MQTT_EVENT_DISCONNECTED");
        connected = false;
        break;

    case MQTT_EVENT_SUBSCRIBED:
        ESP_LOGI(TAG, "MQTT_EVENT_SUBSCRIBED, msg_id=%d", event->msg_id);
        msg_id = esp_mqtt_client_publish(client, "/topic/qos0", "data", 0, 0, 0);
        ESP_LOGI(TAG, "sent publish successful, msg_id=%d", msg_id);
        break;
    case MQTT_EVENT_UNSUBSCRIBED:
        ESP_LOGI(TAG, "MQTT_EVENT_UNSUBSCRIBED, msg_id=%d", event->msg_id);
        break;
    case MQTT_EVENT_PUBLISHED:
        ESP_LOGI(TAG, "MQTT_EVENT_PUBLISHED, msg_id=%d", event->msg_id);
        break;
    case MQTT_EVENT_DATA:
        ESP_LOGI(TAG, "MQTT_EVENT_DATA");
        printf("TOPIC=%.*s\r\n", event->topic_len, event->topic);
```

```
        printf("DATA=%.*s\r\n", event->data_len, event->data);
        break;
    case MQTT_EVENT_ERROR:
        ESP_LOGI(TAG, "MQTT_EVENT_ERROR");
        if (event->error_handle->error_type == MQTT_ERROR_TYPE_TCP_TRANSPORT)
        {
            log_error_if_nonzero("reported from esp-tls", event->error_handle-
>esp_tls_last_esp_err);
            log_error_if_nonzero("reported from tls stack", event->error_handle-
>esp_tls_stack_err);
            log_error_if_nonzero("captured as transport's socket errno", event-
>error_handle->esp_transport_sock_errno);
            ESP_LOGI(TAG, "Last errno string (%s)", strerror(event->error_handle-
>esp_transport_sock_errno));
        }
        break;
    case MQTT_EVENT_BEFORE_CONNECT:
        ESP_LOGI(TAG, "MQTT_EVENT_BEFORE_CONNECT");
        break;

    default:
        ESP_LOGI(TAG, "Other event id:%d", event->event_id);
        break;
    }
    return ESP_OK;
}

static void mqtt_event_handler(void *handler_args, esp_event_base_t base, int32_t
event_id, void *event_data)
{
    ESP_LOGD(TAG, "Event dispatched from event loop base=%s, event_id=%d", base,
event_id);
    mqtt_event_handler_cb(event_data);
}

static void mqtt_app_start(void)
{
    esp_mqtt_client_config_t mqtt_cfg = {
        .uri = CONFIG_BROKER_URL,
        .username = MQTT_ACCESS_TOKEN_2,
    };

    ESP_LOGI(TAG, "Conectando ao servidor: "CONFIG_BROKER_URL);

    mqtt_client = esp_mqtt_client_init(&mqtt_cfg);
    esp_mqtt_client_register_event(mqtt_client, ESP_EVENT_ANY_ID,
mqtt_event_handler, mqtt_client);
    esp_mqtt_client_start(mqtt_client);
}

int mqtt_send(char *jsonString)
{
    return esp_mqtt_client_publish(mqtt_client, MQTT_GATEWAY_TELEMETRY_TOPIC,
jsonString, 0, 2, 0);
}

void mqtt_init(void)
{
    ESP_LOGI(TAG, "[APP] Startup..");
    ESP_LOGI(TAG, "[APP] Free memory: %d bytes", esp_get_free_heap_size());
    ESP_LOGI(TAG, "[APP] IDF version: %s", esp_get_idf_version());

    esp_log_level_set("MQTT_CLIENT", ESP_LOG_VERBOSE);
    esp_log_level_set("MQTT_EXAMPLE", ESP_LOG_VERBOSE);
    esp_log_level_set("TRANSPORT_TCP", ESP_LOG_VERBOSE);
    esp_log_level_set("TRANSPORT_SSL", ESP_LOG_VERBOSE);
```

```
    esp_log_level_set("TRANSPORT", ESP_LOG_VERBOSE);
    esp_log_level_set("OUTBOX", ESP_LOG_VERBOSE);

    ESP_ERROR_CHECK(nvs_flash_init());
    ESP_ERROR_CHECK(esp_netif_init());
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    ESP_ERROR_CHECK(wifi_connect());

    mqtt_app_start();

    while (!connected)
        vTaskDelay(pdMS_TO_TICKS(1));
}
```

Apêndice G - Visada do Nó de Monitoramento em direção a Ponte de Comunicação

Apêndice H - Visada da Ponte de Comunicação em direção ao Nó de monitoramento