**Universidade de Brasília**
Departamento de Ciência da Computação
Teleinformática e Redes 2

# Sistema de Monitoramento de Salas via LoRa

Trabalho Final - 1a Entrega Parcial

Adriele Evellen Alves de Abreu 20/2042785
Fernando Nunes de Freitas 22/2014661
Samuel Andrade de Matos 17/0155943

Professor:
Jacir Luiz Bordim

**21 de outubro de 2025**

# 1 Introdução

## 1.1 Contextualização

O monitoramento contínuo das condições ambientais em salas de servidores e equipamentos de rede é crucial para garantir a integridade, o desempenho e a longevidade dos dispositivos. Variações de temperatura e umidade, bem como o acúmulo de poeira, podem levar a falhas de hardware, interrupções de serviço e custos de manutenção elevados. Soluções de monitoramento tradicionais muitas vezes dependem de infraestrutura de rede cabeada (Ethernet) ou Wi-Fi, o que pode ser inviável em locais de difícil acesso ou com cobertura de rede limitada.

## 1.2 Objetivos do Projeto

O objetivo deste trabalho é projetar e implementar um sistema de monitoramento distribuído, robusto e de baixo custo, que supere as limitações das redes tradicionais. Para isso, o sistema emprega a tecnologia LoRa (Long Range) para a comunicação entre os nós sensores e um gateway central.

O sistema foi dividido em três componentes principais, alinhados às camadas de rede:

1. **Nós Sensores (Cliente LoRa):** Responsáveis pela coleta de dados (temperatura, umidade, poeira) e transmissão via LoRa (Camadas Física e de Enlace).
2. **Gateway (LoRa → IP):** Um dispositivo que atua como ponte, recebendo pacotes LoRa e retransmitindo-os pela rede IP da universidade (Camadas de Rede e Transporte).
3. **Servidor e Dashboard:** Uma aplicação central que recebe os dados do gateway, armazena-os em um banco de dados e os apresenta em uma interface web (Camada de Aplicação).

Um requisito fundamental do projeto foi o desenvolvimento do servidor e gateway utilizando, preferencialmente, bibliotecas padrão do Python (`socket`, `http.server`, `threading`, `sqlite3`, `queue`), a fim de demonstrar o entendimento dos protocolos e primitivas de rede.

# 2 Arquitetura do Sistema

A arquitetura do sistema segue um modelo de três camadas, com responsabilidades bem definidas para cada componente.

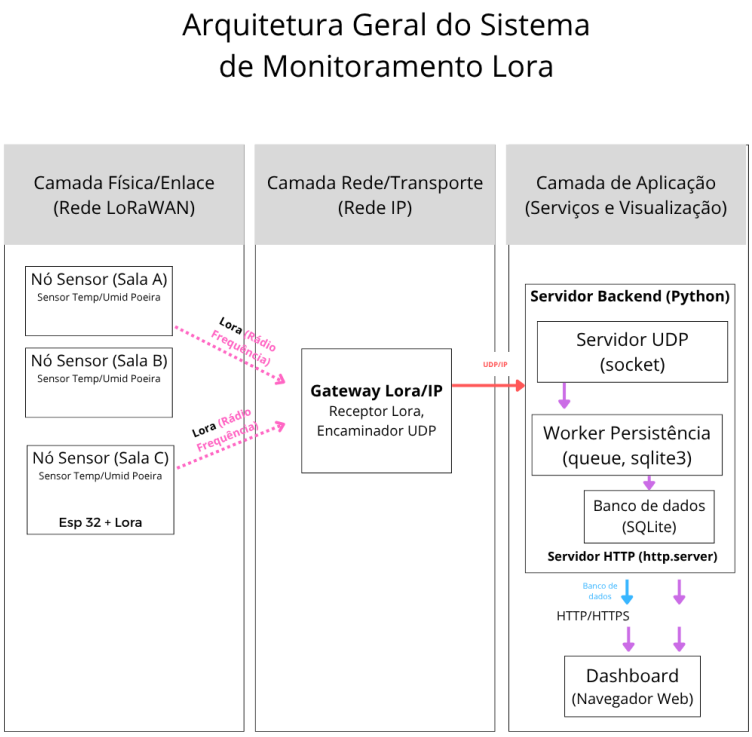


Figura 1: Arquitetura Geral do Sistema de Monitoramento Lora

## 2.1 Fluxo de Comunicação do Sistema

1. Os **Nós Sensores**, posicionados nas salas de equipamentos, coletam dados de **temperatura**, **umidade** e **poeira**.
2. Esses dados são encapsulados em pacotes LoRa e transmitidos via rádio (Camada Física/Enlace) para o **Gateway**.
3. O **Gateway** recebe os pacotes LoRa, extrai os dados e os reempacota em um *payload* JSON.
4. O **Gateway** então envia esse *payload* JSON através da rede IP da universidade para o **Servidor Backend**, utilizando o protocolo **UDP** na porta 9001 (Camada de Rede/Transporte).
5. O **Servidor Backend** (Python) possui uma *thread* `servidor_udp` que recebe os datagramas UDP, decodifica o JSON e coloca os dados em uma fila interna (`queue`).
6. Uma segunda *thread* (`worker_persistencia`) consome os dados da fila e os salva de forma assíncrona no **Banco de Dados** (SQLite).
7. Paralelamente, o **Servidor Backend** expõe uma interface web utilizando o módulo `http.server` (Camada de Aplicação).
8. O **Usuário Final**, através de um **Dashboard** (navegador web), realiza requisições HTTP (`GET`) para o servidor.
9. O servidor responde a essas requisições:
   - `GET /last`: busca o último dado armazenado na memória (tempo real);
   - `GET /all`: consulta o histórico no banco de dados;
   - `GET /`: serve os arquivos `index.html`, `style.css` e `script.js`.

## 2.2 Mapeamento das Camadas

O fluxo de dados do sistema demonstra a interação entre as camadas do modelo OSI/TCP-IP:

- **Camada Física/Enlace:** A comunicação entre os múltiplos Nós Sensores (ESP32 + SX1278) e o Gateway é realizada via rádio frequência utilizando o protocolo LoRa. Esta camada é responsável pela modulação do sinal e pelo endereçamento físico dos dispositivos.
- **Camada de Rede/Transporte:** O Gateway atua como um roteador de borda. Ao receber um pacote LoRa, ele o encapsula em um payload JSON e o envia através da rede IP local. Para esta comunicação, foi escolhido o protocolo **UDP** (User Datagram Protocol) devido à sua leveza e adequação para envio periódico de dados de sensores, onde a perda eventual de um pacote não é catastrófica. O Servidor Backend escuta em uma porta UDP específica para receber esses dados.
- **Camada de Aplicação:** Esta camada possui duas frentes:
  1. **Formato de Mensagem (JSON):** Os dados são serializados em formato JSON, um padrão leve e legível para troca de informações.
  2. **Servidor e Dashboard:** O Servidor Backend (Python) processa o JSON recebido, armazena no `sqlite3` e o disponibiliza através de uma API HTTP. O Dashboard (HTML/JS) consome essa API via `fetch` e exibe os dados ao usuário.

## 2.3 Formato da Mensagem (Payload)

Para garantir a interoperabilidade entre o Gateway e o Servidor, foi definido um formato de mensagem JSON padronizado. O módulo `payload.py` é responsável por criar este formato.

```
{
    "sala": "Servidor",
    "timestamp": 1678886400,
    "temperatura": 25.4,
    "umidade": 55.1,
    "poeira": 15,
    "seq": 101
}
```

Listing 1: Exemplo de payload gerado (payload.py)

# 3 Tecnologias Utilizadas

## 3.1 Hardware (Nós Sensores e Gateway)

Conforme descrito no `README.md` do projeto, os seguintes componentes de hardware foram utilizados para a montagem dos nós sensores e do gateway:

- **Microcontrolador:** 2x ESP32-S3R8
- **Comunicação LoRa:** 2x LoRa SX1278
- **Sensores:**
  - SHT40/41 (Temperatura e Umidade) - Comunicação via I2C
  - DSM501A (Poeira) - Comunicação via PWM

## 3.2 Software e Protocolos

- **Backend (Servidor):** Python 3.10+
- **Bibliotecas Padrão (Python):**
  - `socket`: Para criação do servidor UDP.
  - `http.server`: Para servir o dashboard web e a API JSON.
  - `sqlite3`: Para armazenamento persistente dos dados.
  - `threading`: Para permitir que o servidor UDP, o servidor HTTP e o worker de persistência rodem concorrentemente.
  - `queue`: Para desacoplar o recebimento rápido de dados (UDP) da escrita mais lenta no banco de dados.
  - `json`, `time`, `logging`, `os`.
- **Frontend (Dashboard):** HTML5, CSS3 e JavaScript (ES6+), sem frameworks.
- **Protocolos de Comunicação:** LoRa, UDP, IP, HTTP.
- **Formato de Dados:** JSON.

# 4 Implementação Detalhada

A implementação foi modularizada em três partes principais: o simulador do gateway, o servidor backend e o dashboard web.

## 4.1 Gateway (Simulador)

Para o desenvolvimento e teste do servidor (Entregável 1), foi criado um simulador de gateway em Python (`gateway_udp_sim.py`). Este script simula o comportamento do hardware real: ele gera dados ambientais falsos e os envia periodicamente para o servidor via UDP.

O núcleo do simulador utiliza a biblioteca `socket` para criar um socket UDP e enviar os dados formatados pelo módulo `payload.py`.

```
import socket, random, time
from config import UDP_HOST, UDP_PORT, PERIODO_S, SALA
from payload import build_payload


def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    seq = 0
    try:
        while True:
            temp, umid, poeira = amostra_fake()
            pkt = build_payload(SALA, seq, temp, umid, poeira)
            sock.sendto(pkt, (UDP_HOST, UDP_PORT))
            print(f"Enviado seq={seq} ...")
            seq = (seq + 1) & 0x7fffffff
            time.sleep(PERIODO_S)
```

```
    except KeyboardInterrupt:
        pass
    finally:
        sock.close()
```

Listing 2: Trecho principal do gateway_udp_sim.py

## 4.2 Servidor Backend

O servidor backend é o cérebro do sistema, executado por `app_run.py`. Ele inicializa e coordena três threads principais, garantindo o paralelismo das tarefas.

### 4.2.1 Arquitetura Multi-Thread

O script `app_run.py` demonstra o uso de `threading` para executar tarefas concorrentes:

1. **Worker de Persistência:** Uma thread (`worker_persistencia`) que consome dados de uma fila e os salva no banco de dados.
2. **Servidor UDP:** Uma thread (`servidor_udp`) que escuta por pacotes de dados dos gateways.
3. **Servidor HTTP:** A thread principal (`start_http`) que serve o dashboard e a API.

### 4.2.2 Recepção e Enfileiramento (udp_server.py)

O servidor UDP (`servidor_udp.py`) é responsável por escutar na porta 9001. Ao receber um datagrama, ele:

1. Decodifica a mensagem (JSON).
2. Atualiza o estado em memória (`state.py`) com o último valor recebido daquela sala. Isso é feito com um `threading.Lock` (`lock_ultimo`) para garantir a segurança em ambiente concorrente.
3. Coloca o documento (dicionário Python) em uma `queue.Queue` (`state.fila`).

O uso da `Queue` é uma decisão de design crucial: ela desacopla o processo de recepção (I/O de rede, muito rápido) do processo de escrita em disco (I/O de banco de dados, mais lento), evitando que o servidor UDP bloqueie e perca pacotes.

### 4.2.3 Persistência (storage.py e worker_persistencia)

O módulo `storage.py` gerencia a interação com o banco de dados `sqlite3`. Ele define o schema (DDL) e as funções `salvar(doc)` e `get_all(limit)`.

A função `worker_persistencia` (definida em `udp_server.py`) roda em uma thread separada, em um loop infinito, executando `fila.get()` (que bloqueia até que um item esteja disponível) e, em seguida, chamando `salvar(doc)`.

```
DDL = """
CREATE TABLE IF NOT EXISTS medidas (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  sala TEXT NOT NULL,
  ts INTEGER NOT NULL,
  temperatura REAL NOT NULL,
  umidade REAL NOT NULL,
  poeira INTEGER NOT NULL,
  seq INTEGER NOT NULL
);
CREATE INDEX IF NOT EXISTS idx_medidas_sala_ts ON medidas(sala, ts);
"""
```

Listing 3: Definição da tabela (storage.py)

## 4.3 Dashboard Web (Frontend)

O dashboard é servido pelo próprio backend Python através do módulo `http_dashboard.py`, que utiliza `http.server.BaseHTTPRequestHandler`.

#### 4.3.1 Servidor HTTP e API

O handler HTTP foi customizado para responder a rotas específicas:

- `/`: Serve o `index.html`.
- `/style.css` e `/script.js`: Servem os assets estáticos.
- `/last`: Uma API JSON que retorna o último estado conhecido de todas as salas (lido da variável `state.ultimo`).
- `/all`: Uma API JSON que retorna os últimos 200 registros do banco de dados (chamando `storage.get_all(200)`).

Isso cumpre o requisito de implementar o servidor sem frameworks externos como Flask ou Django.

#### 4.3.2 Interface do Usuário (script.js)

O frontend (`script.js`) utiliza JavaScript moderno (assíncrono com `async/await` e `fetch`) para consumir a API:

- **Dados em Tempo Real:** A função `tick()` é chamada a cada 2 segundos. Ela busca dados em `/last` e atualiza a tabela "live-data-table".
- **Histórico:** A função `loadHistory()` é chamada quando o usuário clica no botão "Mostrar Histórico". Ela busca dados em `/all` e preenche a tabela de histórico.

# 5 Resultados e Demonstração

O sistema foi executado com sucesso em um ambiente de teste. O fluxo de dados pôde ser observado desde a simulação no gateway até a exibição no dashboard web.

## 5.1 Execução do Sistema

Para executar o protótipo, os seguintes comandos são utilizados em terminais separados:

1. **Iniciar o Servidor Backend (UDP e HTTP):**

   ```
   python3 -m servidor_backend.app_run
   ```

   O servidor inicia e informa que está ouvindo na porta UDP 9001 e servindo o dashboard em `http://localhost:8000`.

2. **Iniciar o Simulador do Gateway:**

   ```
   python3 gateway_lora/gateway_udp_sim.py
   ```

   O simulador começa a enviar pacotes UDP a cada 5 segundos (foi utilizado 5 segundos para fins de teste, quando for passado para os sensores o tempo será drasticamente alterado, tendo em vista o alto consumo energético que teria se o sensor enviasse os dados a cada 5 segundos). O terminal do servidor começa a registrar o recebimento das mensagens.

3. **Acessar o Dashboard:** Abrindo `http://localhost:8000` em um navegador, o dashboard é exibido.

## 5.2 Análise do Dashboard

O dashboard apresentou o comportamento esperado:

- A tabela "Dados em Tempo Real"foi atualizada automaticamente a cada 2 segundos, refletindo os últimos dados enviados pelo simulador.
- Ao clicar em "Mostrar Histórico", a tabela de histórico foi preenchida com os dados persistidos no banco de dados `sqlite3`, demonstrando o funcionamento do fluxo de persistência (UDP → Queue → Worker → SQLite → API `/all`).

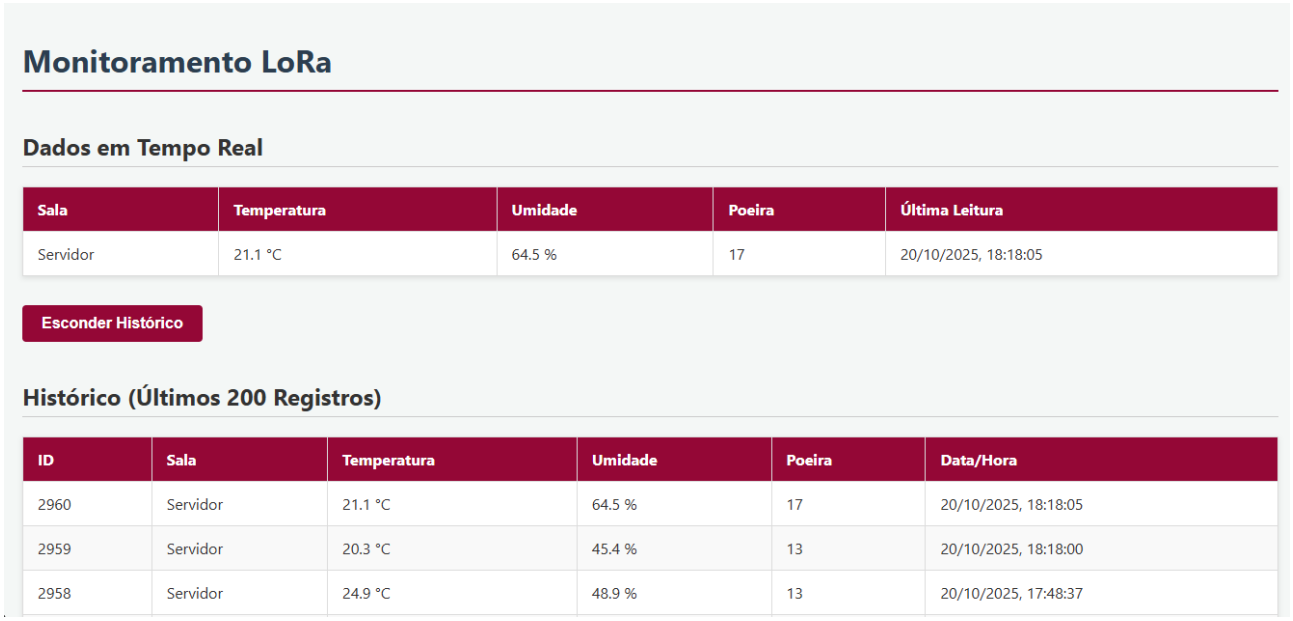


Figura 2: Dashboard web exibindo dados simulados.