Relatório do trabalho de Monitoramento Inteligente de Carga

Carlos **Miguel** do Carmo **do Ó** Gomes
Daniel Almeida **Castro**
Gabriel Ramos **Belizario** Rosas

November 12, 2025

Abstract

Este relatório detalha a arquitetura e implementação de um sistema de telemetria para monitoramento de luminosidade. O sistema utiliza o protocolo UDP para comunicação entre um Cliente em C++ (simulando um sistema embarcado) e um Servidor em Python (Host), que exibe os dados em tempo real em uma interface gráfica. A escolha do UDP prioriza a baixa latência e a natureza de ”fogo e esquece” da telemetria de sensores, ideal para dados de status.

1 Arquitetura e Componentes

1.1 Diagrama de Classes

O diagrama abaixo ilustra o fluxo de dados do sensor para o host que define a estrutura e as relações entre os módulos de software do Cliente (C++) e do Servidor (Python) e a comunicação via datagramas UDP.

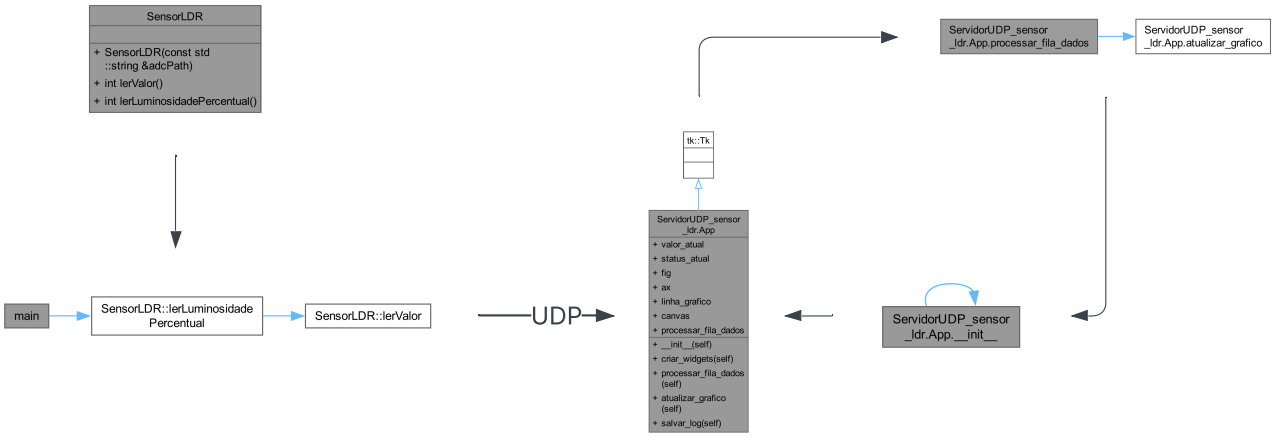

Figure 1: Diagrama de Execução do Cliente C++ (Sistema Embarcado) e do Servidor Python.

1.2 Descrição do Sensor e Funcionamento

O sistema utiliza um Sensor de Luminosidade (LDR, Light-Dependent Resistor) para medir a intensidade da luz ambiente, simulando o comportamento de um sistema embarcado.

1.2.1 Princípio de Funcionamento do LDR

O LDR é um componente cuja resistência elétrica varia em função da intensidade de luz incidente.

- **Escuro:** Alta resistência, resultando em uma leitura de ADC (Analog-to-Digital Converter) próxima de 0.
- **Claro:** Baixa resistência, resultando em uma leitura de ADC próxima do valor máximo ($\sim$4095 para um ADC de 12 bits).

No sistema Cliente C++, a leitura do sensor é convertida de um valor cru para uma métrica de luminosidade normalizada.

1.2.2 Normalização de Dados

A leitura digital bruta do ADC é mapeada com uma linearização logarítmica para uma porcentagem (0% a 100%) antes de ser encapsulada no datagrama UDP. Essa normalização é essencial para que o Servidor Python receba um valor universalmente interpretável para exibição gráfica e log.

2 Fluxogramas de Execução

2.1 Fluxograma Cliente C++

O fluxo demonstra o ciclo de leitura, processamento e transmissão periódica do módulo embarcado.

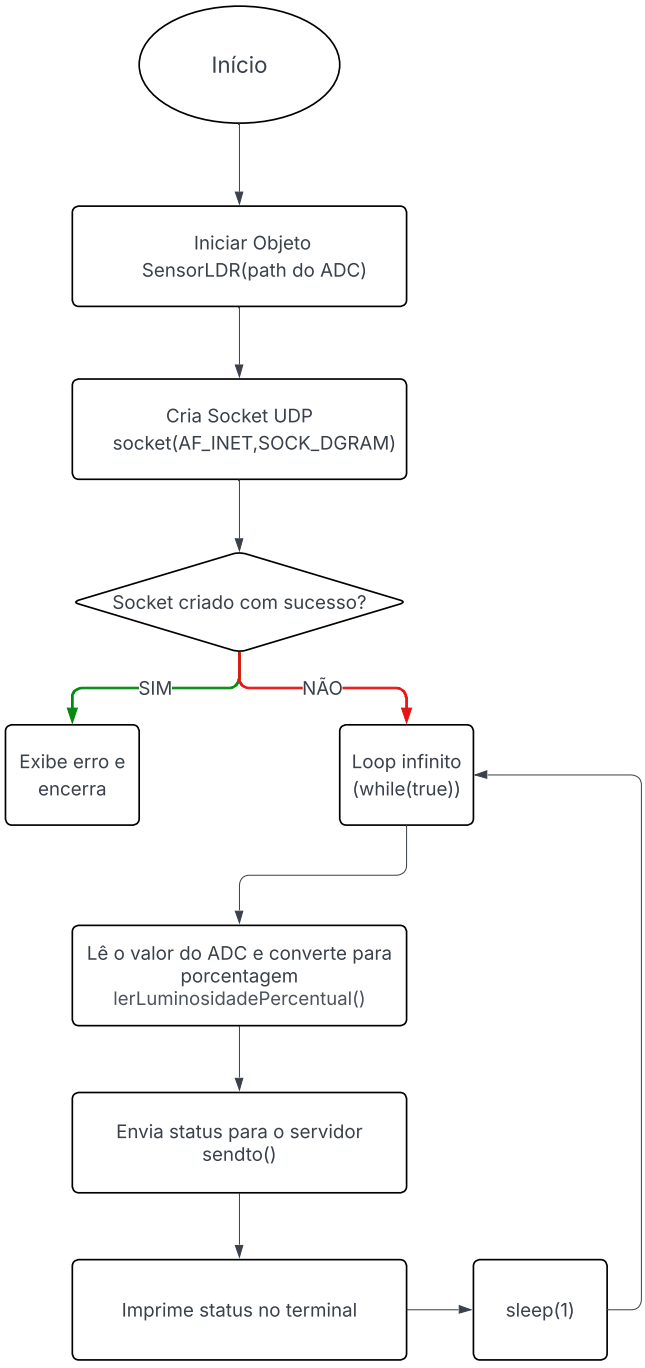

Figure 2: Fluxograma de Execução do Cliente C++ (Sistema Embarcado).

2.2 Fluxograma Servidor Python

O fluxo do servidor destaca o uso de múltiplas threads para separar a escuta de rede da atualização da GUI.

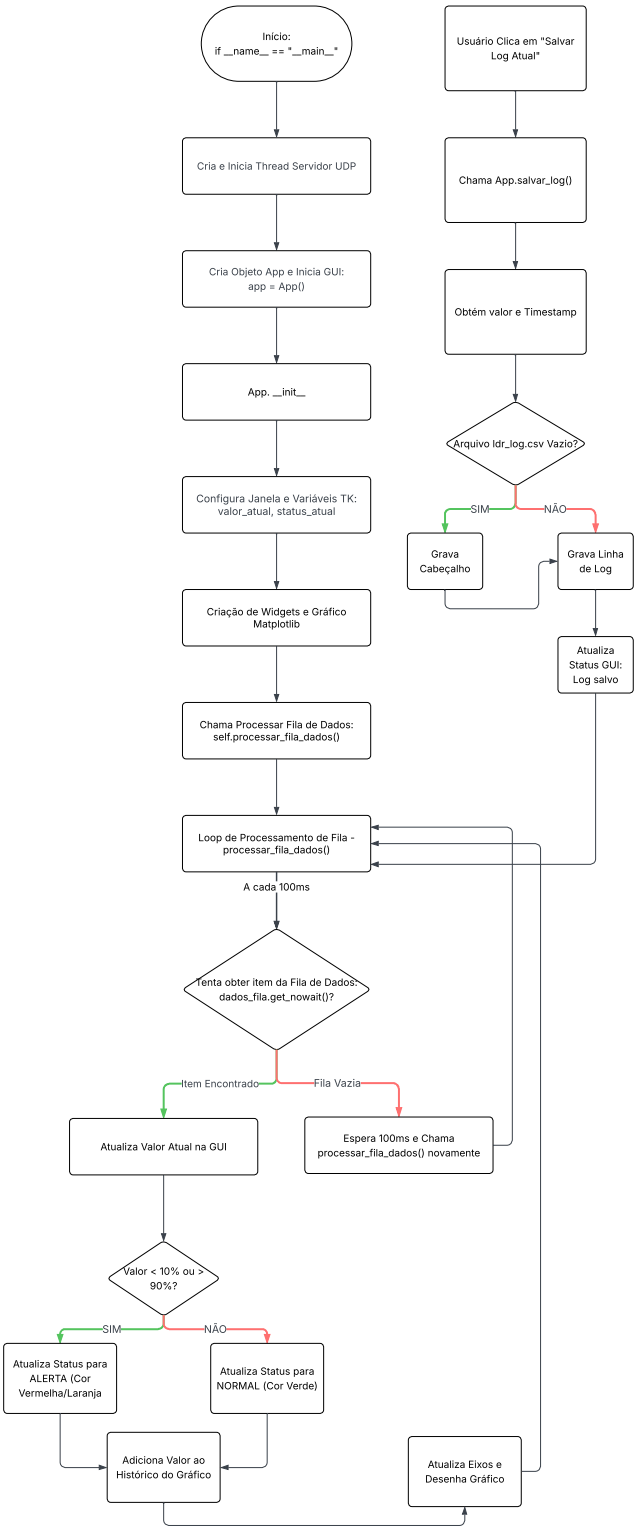

Figure 3: Fluxograma de Execução do Servidor Python (Host de Monitoramento).

3 Instruções Técnicas

3.1 Instruções de Compilação e Uso (Cliente C++)

O Cliente C++ foi projetado para ser compilado em um ambiente Linux (simulando um sistema embarcado) e envia dados para o endereço IP `192.168.42.10` na porta `8080`.

3.2 Compilar o Código (Cross-compilation)

O código C++ precisa ser compilado para a arquitetura **ARM** (compilação cruzada) usando uma *toolchain* específica para o kit de desenvolvimento DK1.

1. **Baixar a Toolchain:** Baixe o arquivo `arm-buildroot-linux-gnueabihf_sdk-buildroot.tar.gz` neste link. Esta toolchain permitirá a compilação cruzada para o kit de desenvolvimento DK1.
2. **Instalar a Toolchain:** Extraia o arquivo baixado no seu ambiente de desenvolvimento usando o seguinte comando `bash`:

```
tar -xvf arm-buildroot-linux-gnueabihf_sdk-buildroot.tar.gz
```

Listing 1: Extrair Toolchain

3. **Compilação Cruzada:** Execute o comando abaixo, que utiliza o `g++` da toolchain para gerar o executável `clienteUDP_sensor_ldr_arm` compatível com a arquitetura da placa STM32MP1:

```
arm-linux-gnueabihf-g++ -o clienteUDP_sensor_ldr_arm
    clienteUDP_sensor_ldr.cpp
```

Listing 2: Compilação para ARM

3.3 Executar na Placa STM32MP1

Após a compilação cruzada, o arquivo executável deve ser transferido para a placa STM32MP1 e executado.

1. **Transferir o Executável:** Transfira o arquivo executável `clienteUDP_sensor_ldr_arm` para a placa via `scp`.
2. **Permissão de Execução:** No terminal da placa, conceda permissão de execução ao arquivo:

```
chmod +x clienteUDP_sensor_ldr_arm
```

Listing 3: Conceder Permissão

3. **Executar o Programa:** Execute o programa no terminal da placa. O programa começará a exibir as leituras de luminosidade no terminal.

```
./clienteUDP_sensor_ldr_arm
```

Listing 4: Execução

3.4 Instruções de Uso (Servidor Python)

O Servidor Python requer as bibliotecas `tkinter` (GUI) e `matplotlib` (Gráficos) para funcionar.

1. **Dependências:** Instale a biblioteca `matplotlib` se não estiver disponível:

```
pip install matplotlib
```

2. **Execução (No Host):** O script deve ser executado usando o interpretador Python 3:

```
python3 servidorUDP_sensor_ldr.py
```

3. **Rede e Firewall:** O servidor escuta na porta `8080`. É **crucial** que o firewall do Host de monitoramento esteja configurado para permitir o tráfego UDP de entrada nesta porta, caso contrário, os dados do Cliente C++ serão bloqueados.

4 Resultados e Análise

4.1 Capturas de Tela da Interface

Esta seção contém as capturas de tela da GUI do Servidor Python em funcionamento, demonstrando o gráfico de histórico em tempo real e o medidor de valor atual.

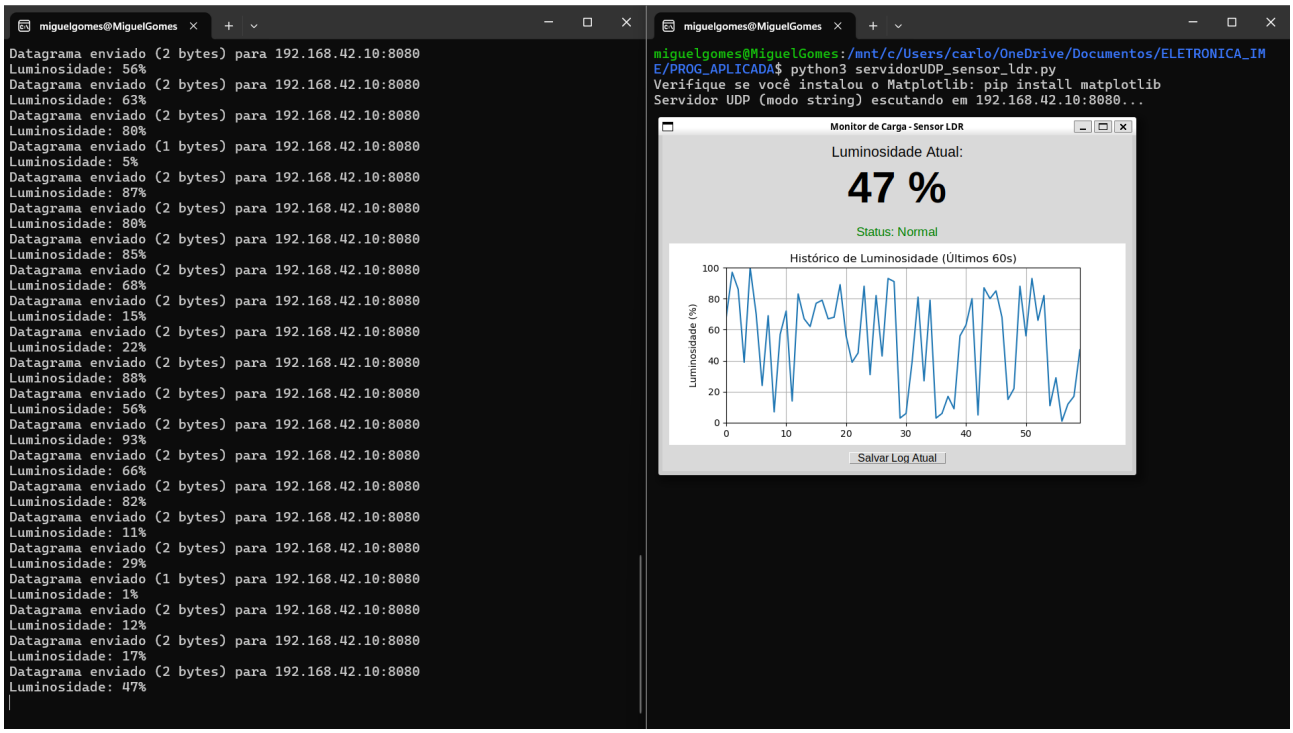

Figure 4: Interface gráfica do Servidor Python exibindo o monitoramento de luminosidade

4.2 Análise dos Valores Medidos

4.2.1 Comportamento em Variação de Luminosidade

O Cliente C++ envia dados a cada 1 segundo. Quando há uma mudança abrupta no ambiente (e.g., luz acesa/apagada), o valor de luminosidade (0% a 100%) deve ser refletido no gráfico do Servidor

Python com uma latência mínima. A simulação no cliente é projetada para variar o valor do sensor de maneira realista.

4.2.2 Análise de Perda de Pacotes

O uso do protocolo UDP (*User Datagram Protocol*) implica que a entrega dos pacotes **não é garantida**. Para aplicações de telemetria em tempo real, onde a informação mais recente é mais valiosa do que a confirmação de uma leitura antiga, o UDP é o protocolo ideal. O Servidor Python lida com a possível perda de datagramas simplesmente mantendo o último valor válido e esperando pelo próximo datagrama, sem implementar mecanismos complexos de retransmissão. A taxa de perda de pacotes será baixa em uma rede local estável.

A Apêndice A: Código-Fonte Cliente C++ (`clienteUDP_sensor_ldr.cpp`)

```
/**
* @file clienteUDP_sensor_ldr.cpp
* @brief Implementação de leitura de luminosidade utilizando um sensor LDR em um sistema Linux embarcado.
* * @author Miguel Gomes
* @date 22 de Outubro de 2025
* * @details Este programa lê valores do ADC exportados no sysfs e converte a resistência do LDR
* em uma estimativa percentual de luminosidade (0% = escuro, 100% = claro).
* O valor lido é então enviado via protocolo UDP (datagrama) para o servidor
* rodando no endereço SERVER_IP (Host Windows/WSL) na porta 8080.
*/

#include <iostream>
#include <fstream>
#include <cstring>
#include <unistd.h>
#include <cmath>
#include <sys/socket.h> // Funções de socket (POSIX)
#include <arpa/inet.h> // Funções de conversão de endereço (inet_pton)
#include <string> // Necessário para std::string e std::to_string

using namespace std;

/** @def SERVER_IP
* @brief Endereço IP do servidor (Host Windows/WSL).
*/
#define SERVER_IP "192.168.42.10"

/** @def PORT
* @brief Porta UDP do servidor.
*/
#define PORT 8080

/** @def BUFFER_SIZE
* @brief Tamanho máximo do buffer de comunicação.
*/
#define BUFFER_SIZE 1024

/**
* @class SensorLDR
* @brief Classe para leitura e cálculo de luminosidade a partir de um LDR.
*
* @details A classe realiza a leitura de valores crus do ADC, calcula a resistência do LDR
```

```
43      * e mapeia para uma escala percentual de luminosidade, utilizando a lei de potência
44      * que relaciona a resistência com a intensidade luminosa (logarítmica).
45      */
46      class SensorLDR {
47        private:
48        /**< Caminho do arquivo sysfs do ADC (ex.: `/sys/bus/iio/devices/.../
              in_voltage13_raw`). */
49        std::string path;
50
51        /**< Resistência aproximada do LDR em ambiente claro (ohms). */
52        float R_CLARO = 146*1e3;
53
54        /**< Resistência aproximada do LDR em ambiente escuro (ohms). */
55        float R_ESCURO = 5*1e6;
56
57        /**< Valor máximo do ADC (12 bits). */
58        const float ADC_MAX = 4095.0;
59
60        /**< Resistência fixa usada no divisor resistivo (ohms). */
61        const float R_FIXO = 10000.0;
62
63        public:
64        /**
65        * @brief Construtor da classe SensorLDR.
66        * @param adcPath Caminho para o arquivo sysfs do ADC.
67        */
68        SensorLDR(const std::string& adcPath) {
69          path = adcPath;
70        }
71
72        /**
73        * @brief Lê o valor cru do ADC.
74        *
75        * @details A leitura é feita diretamente do arquivo de dispositivo (sysfs)
              configurado no path.
76        *
77        * @return Valor inteiro lido diretamente do ADC.
78        */
79        int lerValor() {
80          std::ifstream file(path);
81          int valor = 0;
82          if (file.is_open()) {
83            file >> valor;
84            file.close();
85          } else {
86            // Em um sistema embarcado real, aqui deve haver um tratamento de erro mais
                  robusto.
87            cerr << "Erro: Nao foi possivel abrir o arquivo ADC em " << path << endl;
88          }
89          return valor;
90        }
91
92        /**
93        * @brief Calcula a luminosidade em percentual com base no valor lido do ADC.
94        *
95        * @details O cálculo utiliza a fórmula do divisor de tensão para obter a R_LDR,
96        * e então mapeia a resistência (em escala logarítmica) para uma porcentagem (0-100)
              .
97        *
98        * @return Luminosidade percentual (0 a 100).
99        */
100       int lerLuminosidadePercentual() {
101         int valor = lerValor();
```

```
102         // Fórmula do divisor de tensão: R_LDR = R_FIXO * (ADC_MAX - V_LDR) / V_LDR
103         // Onde V_LDR é valor lido do ADC (ADC_MAX é a tensão de referência)
104         float r_ldr = R_FIXO * (ADC_MAX - valor) / valor;
105
106         float log_r_ldr = log(r_ldr);
107         float log_r_claro = log(R_CLARO);
108         float log_r_escuro = log(R_ESCURO);
109
110         // Limita o valor para 0%
111         if (log_r_ldr > log_r_escuro) {
112           return 0;
113         }
114         // Limita o valor para 100%
115         if (log_r_ldr < log_r_claro) {
116           return 100;
117         }
118
119         // Mapeamento linear na escala logarítmica
120         float porcentagem = 100.0 * (log_r_escuro - log_r_ldr) / (log_r_escuro -
                log_r_claro);
121
122         return static_cast<int>(porcentagem);
123       }
124     };
125
126     /**
127     * @brief Função principal.
128     *
129     * @details Configura o socket UDP para enviar dados para o servidor
            192.168.42.10:8080.
130     * Cria um objeto SensorLDR, lê continuamente a luminosidade, converte o valor inteiro
131     * para string e envia o datagrama para o servidor a cada segundo.
132     *
133     * @return 0 em caso de execução normal.
134     */
135     int main() {
136       // Inicializa o sensor LDR com o caminho do arquivo ADC no sysfs da placa
137       SensorLDR ldr("/sys/bus/iio/devices/iio:device0/in_voltage13_raw");
138
139       int client_socket;
140       struct sockaddr_in server_addr;
141       char buffer[BUFFER_SIZE];
142
143       // 1. Criar o Socket
144       // AF_INET: Endereços IPv4
145       // SOCK_DGRAM: Protocolo UDP (datagrama, sem conexão)
146       client_socket = socket(AF_INET, SOCK_DGRAM, 0);
147       if (client_socket < 0) {
148         perror("Erro ao criar o socket UDP do cliente");
149         return -1;
150       }
151
152       // Limpar e configurar o endereço do servidor
153       memset(&server_addr, 0, sizeof(server_addr));
154       server_addr.sin_family = AF_INET;
155       server_addr.sin_port = htons(PORT); // Converte a porta (8080) para a ordem de
              bytes de rede
156
157       // Converter endereço IP de string para formato binário
158       if (inet_pton(AF_INET, SERVER_IP, &server_addr.sin_addr) <= 0) {
159         perror("Endereco IP invalido/nao suportado");
160         close(client_socket);
161         return -1;
```

```
162       }
163
164       cout << "Socket UDP criado com sucesso." << endl;
165
166       /**
167       * @brief Loop principal de leitura e envio.
168       * @details O loop executa leituras e envios a cada 1 segundo.
169       */
170       while (true) {
171         memset(buffer, 0, BUFFER_SIZE);
172
173         // O valor lido do sensor é um INT
174         int val = ldr.lerLuminosidadePercentual();
175
176         // Converte o valor inteiro (int) para uma string C++ (std::string)
177         string val_str = to_string(val);
178
179         // Obtém o ponteiro C-style (const char*) da string para uso na função sendto()
180         const char *message = val_str.c_str();
181
182         // Calcula o tamanho da mensagem (apenas o tamanho da string, sem o terminador
              nulo)
183         size_t message_len = val_str.length();
184
185         /**
186         * @brief Envia o datagrama UDP.
187         * @details O UDP é um protocolo sem conexão e não confiável; a chegada do pacote
188         * não é garantida pelo protocolo e é gerenciada pela aplicação (se necessário).
189         * O uso do UDP prioriza a baixa latência de dados de status em tempo real.
190         * @param client_socket O descritor do socket.
191         * @param message O ponteiro para os dados a serem enviados.
192         * @param message_len O tamanho dos dados.
193         * @param 0 Flags (geralmente 0 para UDP).
194         * @param server_addr O endereço de destino.
195         * @param sizeof(server_addr) O tamanho da estrutura de endereço.
196         */
197         ssize_t bytes_sent = sendto(client_socket, message, message_len, 0, (const struct
                sockaddr *)&server_addr, sizeof(server_addr));
198
199         if (bytes_sent == -1) {
200           perror("Erro ao enviar datagrama");
201         }
202         else{
203           cout << "Datagrama enviado (" << bytes_sent << " bytes) para " << SERVER_IP <<
                ":" << PORT << endl;
204           cout << "Luminosidade: " << message << "%" << std::endl;
205           sleep(1); // Espera 1 segundo antes da próxima leitura/envio
206         }
207       }
208       // O loop é infinito, o código abaixo só seria executado em caso de interrupção
209       // 4. Fechar o Socket
210       close(client_socket);
211       cout << "Socket fechado. Cliente UDP encerrado." << endl;
212       return 0;
213     }
```

Listing 5: Cliente UDP LDR: clienteUDP_sensor_ldr.cpp

B Apêndice B: Código-Fonte Servidor Python (`servidorUDP_sensor_ldr.py`)

```
"""
@file servidorUDP_sensor_ldr.py
@brief Aplicação GUI de Monitoramento em Tempo Real para Sensor LDR via UDP.

Este script Python implementa um monitor gráfico em tempo real para dados
de luminosidade (em porcentagem) recebidos através de pacotes UDP.
Ele usa Tkinter para a interface gráfica e Matplotlib para o gráfico de histórico.
A principal característica é a função iniciar_servidor_udp, que recebe uma
string numérica via UDP e a transforma em um objeto JSON enriquecido
para processamento na GUI.

@author Gabriel Belizario
@date 09 Novembro de 2025
"""

import tkinter as tk
from tkinter import font
import socket
import threading
import json
import queue
from datetime import datetime
from collections import deque

# Importa as bibliotecas do Matplotlib
import matplotlib
matplotlib.use("TkAgg")
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure

# --- Configurações ---
## @def UDP_IP
# O endereço IP local para o servidor UDP.
UDP_IP = "192.168.42.10"  # Alterado para corresponder ao Código 1
## @def UDP_PORT
# A porta UDP para escuta. Deve corresponder à porta configurada no transmissor C++.
UDP_PORT = 8080
## @def HISTORICO_MAX_PONTOS
# Número máximo de pontos de dados a serem mantidos e exibidos no gráfico.
HISTORICO_MAX_PONTOS = 60

# Fila para comunicação entre threads (servidor -> GUI)
## @var dados_fila
# Fila thread-safe para armazenar os dicionários/objetos JSON recebidos.
dados_fila = queue.Queue()
## @var dados_grafico
# Deque para armazenar o histórico de valores para plotagem, com tamanho máximo.
dados_grafico = deque(maxlen=HISTORICO_MAX_PONTOS)

def iniciar_servidor_udp():
"""
@brief Inicia um servidor UDP em um thread separado para receber dados.

Escuta por pacotes UDP contendo uma string numérica. Converte a string
para um número inteiro e a *enriquece* com metadados (ID e unidade)
em um formato de dicionário Python (JSON). O dicionário enriquecido
é então colocado na fila de dados para ser processado pela thread principal (GUI).
O ID do sensor é fixo como "LDR_KY-018" e a unidade como "%".

```

```
60      @note Assume-se que os pacotes UDP contêm apenas a representação string de um inteiro
            .
61      @return Não retorna. Executa em loop infinito.
62      """
63      sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
64      sock.bind((UDP_IP, UDP_PORT))
65      print(f"Servidor UDP (modo string) escutando em {UDP_IP}:{UDP_PORT}...")
66
67      while True:
68      try:
69      # 1. Recebe o datagrama (bytes)
70      data, addr = sock.recvfrom(1024)
71
72      # 2. Converte bytes para string e remove espaços em branco (ex: "75")
73      mensagem_string = data.decode('utf-8').strip()
74
75      # 3. Converte a string (ex: "75") para um número inteiro
76      valor_percentual = int(mensagem_string)
77
78      # 4. TRANSFORMAÇÃO PARA JSON (Dicionário Python)
79      dados_json_enriquecidos = {
80        "id": "LDR_KY-018",# Adiciona o ID do grupo
81        "valor": valor_percentual,
82        "unidade": "%"# Adiciona a unidade
83      }
84
85      # 5. Coloca o objeto JSON (dicionário) na fila
86      dados_fila.put(dados_json_enriquecidos)
87
88      except (ValueError, UnicodeDecodeError):
89      # Erro se a mensagem não for um número (ex: "ola")
90      print(f"Erro: Pacote recebido não é um número válido: {data}")
91      except Exception as e:
92      print(f"Erro no servidor UDP: {e}")
93
94      class App(tk.Tk):
95      """@class App
96      @brief Classe principal que define a Interface Gráfica do Usuário (GUI).
97
98      Gerencia a janela principal, exibe o valor atual, o status de alerta,
99      o gráfico de histórico e a funcionalidade de salvar log.
100     Utiliza o método after() do Tkinter para processar a fila de dados de forma
            assíncrona.
101     """
102     def __init__(self):
103     """
104     @brief Construtor da classe App.
105     Inicializa a janela, variáveis de controle e chama a criação de widgets.
106     """
107     super().__init__()
108     self.title("Monitor de Carga - Sensor LDR")
109     self.geometry("700x500")
110
111     ## @var valor_atual
112     # Variável Tkinter para exibir o valor de luminosidade atual.
113     self.valor_atual = tk.StringVar(value="-- %")
114     ## @var status_atual
115     # Variável Tkinter para exibir a mensagem de status/alerta.
116     self.status_atual = tk.StringVar(value="Aguardando dados...")
117
118     self.criar_widgets()
119     self.processar_fila_dados()
120
```

```
121     def criar_widgets(self):
122     """
123     @brief Configura e empacota todos os componentes da interface.
124     Cria os rótulos, o frame do Matplotlib e o botão de salvar log.
125     """
126     # Frame do Valor Atual
127     frame_valor = tk.Frame(self, pady=10)
128     frame_valor.pack()
129     tk.Label(frame_valor, text="Luminosidade Atual:", font=("Arial", 16)).pack()
130     self.label_valor_atual = tk.Label(frame_valor, textvariable=self.valor_atual, font=("
            Arial", 48, "bold"))
131     self.label_valor_atual.pack()
132 
133     # Frame de Status (Alerta Visual)
134     self.label_status = tk.Label(self, textvariable=self.status_atual, font=("Arial", 14)
            , fg="gray")
135     self.label_status.pack(pady=5)
136 
137     # Frame do Gráfico
138     frame_grafico = tk.Frame(self)
139     frame_grafico.pack(fill=tk.BOTH, expand=True, padx=10)
140 
141     ## @var fig
142     # Figura do Matplotlib para o gráfico.
143     self.fig = Figure(figsize=(6, 3), dpi=100)
144     ## @var ax
145     # Eixo da figura para plotagem.
146     self.ax = self.fig.add_subplot(111)
147     self.ax.set_title("Histórico de Luminosidade (Últimos 60s)")
148     self.ax.set_ylabel("Luminosidade (%)")
149     self.ax.set_ylim(0, 100)
150     self.ax.grid()
151     ## @var linha_grafico
152     # Objeto de linha usado para atualizar o plot de forma eficiente.
153     self.linha_grafico, = self.ax.plot([], [])
154 
155     ## @var canvas
156     # Canvas Tkinter que contém a figura do Matplotlib.
157     self.canvas = FigureCanvasTkAgg(self.fig, master=frame_grafico)
158     self.canvas.draw()
159     self.canvas.get_tk_widget().pack(side=tk.TOP, fill=tk.BOTH, expand=True)
160 
161     # Frame de Ações (Salvar)
162     frame_acoes = tk.Frame(self, pady=10)
163     frame_acoes.pack()
164     btn_salvar = tk.Button(frame_acoes, text="Salvar Log Atual", font=("Arial", 12),
            command=self.salvar_log)
165     btn_salvar.pack()
166 
167 
168     def processar_fila_dados(self):
169     """
170     @brief Verifica a fila de comunicação em busca de novos dados e atualiza a GUI.
171 
172     Utiliza get_nowait() para verificar a fila sem bloquear. Se um dicionário
173     de dados for encontrado, a função atualiza o rótulo de valor, a mensagem
174     de status (alerta) e adiciona o ponto ao gráfico. Chama a si mesma
175     a cada 100ms via self.after().
176     """
177     try:
178     dados = dados_fila.get_nowait()
179 
180     # Esta lógica funciona porque 'dados' é o dicionário
```

```
    # que a função 'iniciar_servidor_udp' criou
    valor = dados.get('valor', 0)

    # 1. Atualiza o Valor Atual
    self.valor_atual.set(f"{valor} %")

    # 2. Atualiza o Alerta Visual
    if valor < 10:
    self.status_atual.set("ALERTA: Escuridão detectada! (Possível violação)")
    self.label_status.config(fg="red")
    elif valor > 90:
    self.status_atual.set("ALERTA: Luz intensa detectada! (Invólucro aberto)")
    self.label_status.config(fg="orange")
    else:
    self.status_atual.set("Status: Normal")
    self.label_status.config(fg="green")

    # 3. Atualiza o Histórico Gráfico
    dados_grafico.append(valor)
    self.atualizar_grafico()

    except queue.Empty:
    # Não havia dados na fila, apenas continua
    pass
    finally:
    self.after(100, self.processar_fila_dados)

    def atualizar_grafico(self):
    """
    @brief Atualiza os dados (eixos X e Y) do gráfico Matplotlib.
    O eixo Y é o histórico de dados_grafico. O eixo X é o índice da lista.
    """
    self.linha_grafico.set_ydata(list(dados_grafico))
    self.linha_grafico.set_xdata(range(len(dados_grafico)))
    self.ax.set_xlim(0, HISTORICO_MAX_PONTOS - 1)
    self.canvas.draw()

    def salvar_log(self):
    """
    @brief Salva o valor atual de luminosidade em um arquivo CSV (ldr_log.csv).
    Inclui um timestamp para cada registro e um cabeçalho se o arquivo for novo.
    """
    valor_log = self.valor_atual.get().split(' ')[0]
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    linha = f"{timestamp},{valor_log}\n"

    try:
    with open("ldr_log.csv", "a") as f:
    if f.tell() == 0:
    f.write("timestamp,luminosidade_percent\n")
    f.write(linha)
    self.status_atual.set(f"Log salvo com sucesso às {timestamp}")
    self.label_status.config(fg="blue")
    except Exception as e:
    self.status_atual.set(f"Erro ao salvar log: {e}")
    self.label_status.config(fg="red")

    # --- Ponto de Entrada Principal ---
    if __name__ == "__main__":
    # 1. Instala a dependência (se ainda não o fez)
    print("Verifique se você instalou o Matplotlib: pip install matplotlib")

    # 2. Inicia o thread do servidor UDP (com a nova lógica)
```

```
244     servidor_thread = threading.Thread(target=iniciar_servidor_udp, daemon=True)
245     servidor_thread.start()
246 
247     # 3. Inicia a aplicação GUI
248     app = App()
249     app.mainloop()
```

Listing 6: Servidor UDP GUI: servidorUDP_sensor_ldr.py