

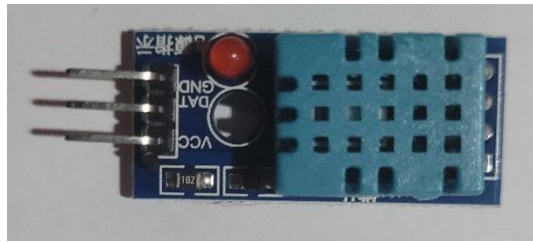
# Realizar práticas de aplicações IoT

Nome: Tiago Lauriano Copelli

## Enunciado 1:

Leitura dos status dos botões da placa Bitdoglab, para que seja visualizado em um servidor: Com base no código apresentado na aula do capítulo 3, da unidade 2, crie um programa para monitorar os status do botão da placa e enviar, a cada 1 segundo os status para um servidor. Além disso, como um desafio extra, acrescente algum sensor e envie a informação desse sensor para o servidor.

Usando o sensor DHT11 para medir temperatura e umidade



Utilizando os pinos 5V, GND e o pino GP8 para os dados.

## Código C Bare Metal

```
#include <string.h>      // Para funções de manipulação de strings como strlen,
                          strncmp

#include "pico/stdlib.h"  // Funções padrão do SDK do Pico (timers, stdio se
                          usado, etc.)

#include "pico/cyw43_arch.h" // Para controle do chip Wi-Fi CYW43 (conexão
                          Wi-Fi, LEDs da placa)

#include "hardware/gpio.h" // Para controle direto dos pinos GPIO

#include "pico/time.h"    // Para funções de temporização (busy_wait_us,
                          time_us_32)

#include <stdio.h>        // Necessário para sprintf/snprintf (formatação de
                          strings)

#include "lwip/tcp.h"     // Para a pilha TCP/IP LwIP (funções do servidor TCP)


// --- Configurações Globais do Projeto ---

#define WIFI_SSID "copelli4"      // nome da sua rede Wi-Fi
#define WIFI_PASS "copelli4"     // senha da rede Wi-Fi
#define PINO_LED_ERRO 11         // Pino GPIO para o LED de
indicação de erro
#define PINO_LED_OK 12           // Pino GPIO para o LED de indicação
de status OK
#define PORTA_TCP 8081           // Porta TCP onde o servidor web
escutará por conexões
#define TIMEOUT_CONEXAO_WIFI_MS 30000 // Tempo máximo (em
milissegundos) para tentar conectar ao Wi-Fi


// --- Configurações dos Pinos GPIO para Sensores ---

#define PINO_BOTAO 5             // Pino GPIO conectado ao botão
#define PINO_DHT11 8            // Pino GPIO conectado ao pino de
dados do sensor DHT11


// --- Constantes Específicas para o Sensor DHT11 ---
```

```

#define MAX_TEMPORIZACOES_DHT 85          // Número máximo de
transições de nível esperadas durante a leitura do DHT11

#define ERRO_TIMEOUT_ESPERA_NIVEL_ALVO ((uint32_t)-1) // Valor de
retorno para erro de timeout ao esperar um nível específico

#define ERRO_TIMEOUT_ESPERA_MUDANCA_NIVEL ((uint32_t)-2) // Valor
de retorno para erro de timeout ao esperar mudança de nível


// --- Estrutura para Armazenar os Dados Lidos do DHT11 ---
typedef struct {
    float temperatura;    // Valor da temperatura em graus Celsius
    float umidade;        // Valor da umidade relativa em porcentagem
    bool checksum_correto; // Indica se o checksum dos dados lidos está correto
} leitura_dht11_t;


// --- HTML com CSS Embutido ---
// Esta string constante define a página web que será enviada ao navegador.
// Inclui CSS para estilização (fundo preto, texto branco, conteúdo centralizado)
// e placeholders (%s, %d, %.1f) que serão substituídos pelos dados dos
sensores.

static const char *g_template_html =
    "<!DOCTYPE html><html><head><title>BitDogLab</title>"
    "<meta http-equiv=\"refresh\" content=\"1\">" // Faz a página recarregar
    automaticamente a cada 1 segundo
    "<style>"
    "body { background-color: #000000; color: #ffffff; font-family: Arial, sans-serif;
    text-align: center; padding-top: 30px; margin-left: 10px; margin-right: 10px; }"
    "h1 { color: #00c0ff; margin-bottom: 25px; }"
    "p { font-size: 1.1em; margin: 10px auto; line-height: 1.5; max-width: 500px; }"
    "span.label { font-weight: bold; color: #a0d8ef; margin-right: 8px; }"
    "span.value-ok { color: #60d060; }"        // Verde para status OK
    "span.value-fail { color: #ff6060; }"      // Vermelho para status Falha
    "span.value-pressed { color: #f0ad4e; font-weight: bold; }" // Laranja para
    botão pressionado

```

```

"</style>"

"</head><body>"

"<h1>Status dos Sensores - BitDogLab</h1>"

"<p><span class=\"label\">Botao (GP%d):</span><span
class=\"%s\">%s</span></p>" // Placeholders para pino, classe CSS e valor
do botão

"<p><span class=\"label\">DHT11 (GP%d):</span><span
class=\"%s\">%s</span></p>" // Placeholders para pino, classe CSS e status
do DHT11

"<p><span class=\"label\">Temperatura:</span>%.1f
&deg;C</p>" // Placeholder para temperatura

"<p><span class=\"label\">Umidade:</span>%.1f %%</p>" //
Placeholder para umidade

"</body></html>";

```

```

// --- Estado Global do Servidor TCP ---

```

```

static struct tcp_pcb *g_pcb_cliente = NULL; // Ponteiro para o Bloco de
Controle de Protocolo (PCB) da conexão do cliente atual

```

```

static struct tcp_pcb *g_pcb_escuta = NULL; // Ponteiro para o PCB do
servidor que está escutando por novas conexões

```

```

// --- Funções Auxiliares ---

```

```

/**

```

```

 * Pisca um LED conectado a um pino GPIO.

```

```

 * * O número do pino GPIO onde o LED está conectado.

```

```

 * O número de vezes que o LED deve piscar.

```

```

 * O intervalo (em milissegundos) entre acender e apagar o LED.

```

```

 */

```

```

void pisca_led(uint pino_led, int vezes, int intervalo_ms) {

```

```

    for (int i = 0; i < vezes; ++i) {

```

```

        gpio_put(pino_led, 1); // Acende o LED

```

```

        sleep_ms(intervalo_ms);

```

```

        gpio_put(pino_led, 0); // Apaga o LED
    }
}

```

```

        sleep_ms(intervalo_ms);
    }
}

/**
 * Função auxiliar para aguardar uma mudança de nível em um pino GPIO ou
 * um timeout.
 * Usada primariamente pela função de leitura do DHT11.
 * O pino GPIO a ser monitorado.
 * esperar_nivel_alto Se true, espera o pino ir para ALTO (1); se false, espera ir
 * para BAIXO (0).
 * Depois, mede quanto tempo o pino permanece nesse estado antes de
 * mudar.
 * timeout_microsegundos Tempo máximo de espera em microssegundos.
 * uint32_t A duração (em microssegundos) que o pino permaneceu no estado
 * `esperar_nivel_alto` antes de mudar,
 * ou um código de erro TIMEOUT se o tempo máximo for atingido.
 */
static uint32_t aguardar_mudanca_nivel_gpio(uint pino_gpio, bool
esperar_nivel_alto, uint32_t timeout_microsegundos) {
    uint32_t inicio_us = time_us_32(); // Pega o tempo atual em microssegundos

    // Loop 1: Espera o pino atingir o estado `esperar_nivel_alto`
    while (gpio_get(pino_gpio) != esperar_nivel_alto) {
        if (time_us_32() - inicio_us > timeout_microsegundos) {
            return ERRO_TIMEOUT_ESPERA_NIVEL_ALVO; // Retorna erro se o
            timeout for atingido
        }
    }

    // Loop 2: O pino atingiu o estado `esperar_nivel_alto`. Agora mede quanto
    tempo ele permanece assim.
    inicio_us = time_us_32(); // Reinicia o contador de tempo
    while (gpio_get(pino_gpio) == esperar_nivel_alto) {

```

```

        if (time_us_32() - inicio_us > timeout_microsegundos) {
            return ERRO_TIMEOUT_ESPERA_MUDANCA_NIVEL; // Retorna erro
            se o timeout for atingido
        }
    }

    return time_us_32() - inicio_us; // Retorna a duração que o pino permaneceu
    no estado `esperar_nivel_alto`
}

```

/\*\*

- \* Lê os dados de temperatura e umidade do sensor DHT11.
  - \* Implementa o protocolo de comunicação "bit-banging" do DHT11.
  - \* resultado Ponteiro para uma struct leitura\_dht11\_t onde os dados lidos serão armazenados.
  - \* true Se a leitura e a verificação do checksum forem bem-sucedidas.
  - \* false Se ocorrer algum erro durante a leitura ou o checksum falhar.
- \*/

```

bool ler_dht11(leitura_dht11_t *resultado) {

```

```

    uint8_t dados_sensor[5] = {0, 0, 0, 0, 0}; // Array para armazenar os 5 bytes
    de dados do DHT11

```

```

    int i;

```

```

    // Inicializa a struct de resultado

```

```

    resultado->checksum_correto = false;

```

```

    resultado->temperatura = 0.0f; // Valor padrão em caso de falha

```

```

    resultado->umidade = 0.0f;    // Valor padrão em caso de falha

```

```

    // Fase 1: Sinal de Start para o DHT11

```

```

    gpio_set_dir(PINO_DHT11, GPIO_OUT); // Configura o pino como saída

```

```

    gpio_put(PINO_DHT11, 0);           // Coloca o pino em nível baixo

```

```

    busy_wait_ms(20);                  // Mantém em baixo por 20ms (mínimo 18ms)

```

```

    gpio_put(PINO_DHT11, 1);           // Coloca o pino em nível alto

```

```

busy_wait_us(40);           // Mantém em alto por 40µs

// Fase 2: Prepara para receber a resposta do DHT11
gpio_set_dir(PINO_DHT11, GPIO_IN); // Configura o pino como entrada

// Fase 3: Leitura da Resposta e dos Dados do DHT11

// O DHT11 deve responder puxando a linha para baixo (~80µs) e depois
para alto (~80µs)

if (aguardar_mudanca_nivel_gpio(PINO_DHT11, false, 120) >=
ERRO_TIMEOUT_ESPERA_NIVEL_ALVO) return false; // Espera resposta
LOW

if (aguardar_mudanca_nivel_gpio(PINO_DHT11, true, 120) >=
ERRO_TIMEOUT_ESPERA_NIVEL_ALVO) return false; // Espera resposta
HIGH

// Loop para ler os 40 bits de dados (5 bytes)
for (i = 0; i < 40; ++i) {
    // Cada bit é precedido por um pulso baixo de ~50µs
    if (aguardar_mudanca_nivel_gpio(PINO_DHT11, false, 70) >=
ERRO_TIMEOUT_ESPERA_NIVEL_ALVO) return false;

    // A duração do pulso alto subsequente determina se o bit é 0 ou 1
    uint32_t duracao_alto = aguardar_mudanca_nivel_gpio(PINO_DHT11,
true, 100);

    if (duracao_alto >= ERRO_TIMEOUT_ESPERA_NIVEL_ALVO) return
false;

    dados_sensor[i / 8] <<= 1; // Desloca o byte para a esquerda para abrir
espaço para o novo bit

    if (duracao_alto > 45) { // Se o pulso alto for maior que ~45µs, considera-
se bit 1 (valor empírico, pode precisar de ajuste)
        dados_sensor[i / 8] |= 1;
    }
}
}

```



```

// Fase 4: Verificação do Checksum

// O checksum é a soma dos 4 primeiros bytes, comparada com o 5º byte.
uint8_t checksum_calculado = (dados_sensor[0] + dados_sensor[1] +
dados_sensor[2] + dados_sensor[3]) & 0xFF;

if (checksum_calculado != dados_sensor[4]) {
    return false; // Checksum falhou
}

// Fase 5: Armazena os dados na struct de resultado
// Para o DHT11, data[1] e data[3] (partes decimais) são geralmente 0.
resultado->umidade = (float)dados_sensor[0];
resultado->temperatura = (float)dados_sensor[2];
resultado->checksum_correto = true; // Leitura bem-sucedida
return true;
}

// --- Funções do Servidor TCP ---
/**
 * Fecha de forma segura a conexão TCP com um cliente.
 * Limpa os callbacks e fecha o PCB.
 * pcb_a_fechar O PCB da conexão do cliente a ser fechada.
 */
static void fechar_conexao_cliente(struct tcp_pcb *pcb_a_fechar) {
    if (pcb_a_fechar) {
        // Remove todos os callbacks e argumentos associados ao PCB
        tcp_arg(pcb_a_fechar, NULL);
        tcp_sent(pcb_a_fechar, NULL);
        tcp_recv(pcb_a_fechar, NULL);
        tcp_err(pcb_a_fechar, NULL);
        tcp_poll(pcb_a_fechar, NULL, 0);
    }
}

```

```

    // Tenta fechar a conexão
    if (tcp_close(pcb_a_fechar) != ERR_OK) {
        tcp_abort(pcb_a_fechar); // Se o fechamento normal falhar, aborta a
conexão
    }
}

// Se o PCB fechado era o cliente ativo global, zera a referência global
if (pcb_a_fechar == g_pcb_cliente) {
    g_pcb_cliente = NULL;
}
}

/**
 * Callback chamado pela pilha LwIP quando ocorre um erro na conexão TCP.
 * arg Argumento definido pelo usuário
 * Código do erro LwIP.
 */
static void server_err_cb(void *arg, err_t err) {
    if (g_pcb_cliente != NULL) { // Se havia um cliente ativo
        g_pcb_cliente = NULL; // Zera a referência, pois a conexão está com
erro
    }

    pisca_led(PINO_LED_ERRO, 3, 150); // Sinaliza o erro de conexão piscando
o LED
}

/**
 * Callback chamado pela pilha LwIP após os dados enviados via tcp_write()
 * serem confirmados (ACKed) pelo cliente.
 * arg Argumento definido pelo usuário.
 * tpcb O PCB da conexão.
 * len O número de bytes confirmados como enviados.

```

```
* err_t ERR_OK se tudo correu bem.
```

```
*/
```

```
static err_t server_sent_cb(void *arg, struct tcp_pcb *tpcb, u16_t len) {  
    pisca_led(PINO_LED_OK, 1, 20); // Pisca LED OK rapidamente para indicar  
    sucesso no envio  
  
    fechar_conexao_cliente(tpcb); // Fecha a conexão, pois a resposta foi  
    enviada  
  
    return ERR_OK;  
}
```

```
/**
```

```
* Callback chamado pela pilha LwIP quando dados são recebidos do cliente.
```

```
* É aqui que a requisição HTTP GET é processada e a página HTML é gerada  
e enviada.
```

```
* arg Argumento definido pelo usuário.
```

```
* tpcb O PCB da conexão.
```

```
* p O buffer (pbuf) contendo os dados recebidos; NULL se o cliente fechou a  
conexão.
```

```
* err Código de erro LwIP.
```

```
* err_t ERR_OK se tudo correu bem.
```

```
*/
```

```
static err_t server_recv_cb(void *arg, struct tcp_pcb *tpcb, struct pbuf *p, err_t  
err) {
```

```
    // Trata erros na recepção ou se o cliente abortou
```

```
    if (err != ERR_OK && err != ERR_ABRT) {
```

```
        if (p) pbuf_free(p); // Libera o buffer se existir
```

```
        fechar_conexao_cliente(tpcb);
```

```
        return err;
```

```
    }
```

```
    // Se p é NULL, o cliente fechou a conexão remotamente
```

```
    if (!p) {
```

```

    fechar_conexao_cliente(tpcb);
    return ERR_OK;
}

// Informa à pilha LwIP que os dados do pbuf foram processados
tcp_recved(tpcb, p->tot_len);

// Verifica se é uma requisição HTTP GET (verificação básica)
if (p->tot_len >= 3 && strncmp((char *)p->payload, "GET", 3) == 0) {
    // Lê o estado atual do botão
    bool botao_esta_pressionado = !gpio_get(PINO_BOTAO); // Pull-up:
    pressionado = nível baixo (0)

    // Lê os dados atuais do sensor DHT11
    leitura_dht11_t leitura_dht_corrente;
    bool leitura_dht_foi_ok = ler_dht11(&leitura_dht_corrente);

    // Prepara strings para os valores e classes CSS dinâmicas
    char str_valor_botao[15]; char str_classe_botao[30];
    sprintf(str_valor_botao, botao_esta_pressionado ? "PRESSIONADO" :
"SOLTO");
    sprintf(str_classe_botao, botao_esta_pressionado ? "value-pressed" :
"value-ok");

    char str_status_dht[30]; char str_classe_dht[30];
    sprintf(str_status_dht, leitura_dht_foi_ok ? "OK" : "Falha na leitura");
    sprintf(str_classe_dht, leitura_dht_foi_ok ? "value-ok" : "value-fail");

    // Monta o corpo do HTML dinamicamente usando o template e os dados
    atuais
    char corpo_html_dinamico[1200]; // Buffer para o corpo HTML

```

```

    int tamanho_necessario_corpo = snprintf(corpo_html_dinamico,
sizeof(corpo_html_dinamico), g_template_html,

        PINO_BOTAO, str_classe_botao, str_valor_botao,

        PINO_DHT11, str_classe_dht, str_status_dht,

        leitura_dht_foi_ok ? leitura_dht_corrente.temperatura : -99.0f, // Usa
-99 se falha

        leitura_dht_foi_ok ? leitura_dht_corrente.umidade : -99.0f); // Usa -
99 se falha


// Verifica se o buffer do corpo HTML foi suficiente
if (tamanho_necessario_corpo >= sizeof(corpo_html_dinamico)) {
    // AVISO: HTML BODY TRUNCADO! O buffer é pequeno demais.
    pisca_led(PINO_LED_ERRO, 5, 100); // Pisca LED de erro
}


// Monta a resposta HTTP completa (cabeçalhos HTTP + corpo HTML)
char resposta_http_completa[1400]; // Buffer para a resposta HTTP
completa

int tamanho_corpo_html = strlen(corpo_html_dinamico);

int tamanho_resposta_http = snprintf(resposta_http_completa,
sizeof(resposta_http_completa),

    "HTTP/1.1 200 OK\r\n"

    "Content-Type: text/html; charset=utf-8\r\n" // Define tipo e
codificação

    "Content-Length: %d\r\n"                // Tamanho do corpo
HTML

    "Connection: close\r\n\r\n"            // Informa que a
conexão será fechada após esta resposta

    "%s",                                // O corpo HTML dinâmico

    tamanho_corpo_html, corpo_html_dinamico);


// Envia a resposta HTTP ao cliente

if (tamanho_resposta_http > 0 && tamanho_resposta_http <
sizeof(resposta_http_completa)) {

```

```

        err_t erro_ao_escrever = tcp_write(tpcb, resposta_http_completa,
tamanho_resposta_http, TCP_WRITE_FLAG_COPY);

        if (erro_ao_escrever == ERR_OK) {

            tcp_output(tpcb); // Tenta enviar os dados imediatamente

            tcp_sent(tpcb, server_sent_cb); // Define callback para quando o
envio for confirmado

        } else {

            pisca_led(PINO_LED_ERRO, 4, 100); // Erro ao tentar escrever para
o socket TCP

            fechar_conexao_cliente(tpcb);

        }

    } else {

        pisca_led(PINO_LED_ERRO, 5, 100); // Erro ao formatar a resposta
HTTP completa ou buffer pequeno

        fechar_conexao_cliente(tpcb);

    }

} else { // Se a requisição não for um GET

    fechar_conexao_cliente(tpcb);

}

pbuf_free(p); // Libera o buffer da requisição recebida
return ERR_OK;
}

```

/\*\*

\* Callback chamado pela pilha LwIP quando uma nova conexão de cliente é aceita

\* no PCB que está escutando.

\* arg Argumento definido pelo usuário.

\* novo\_pcb\_cliente O PCB da nova conexão estabelecida.

\* err Código de erro LwIP.

\* err\_t ERR\_OK se a conexão for aceita, ou um erro LwIP caso contrário.

\*/

```
static err_t server_accept_cb(void *arg, struct tcp_pcb *novo_pcb_cliente, err_t
err) {
    // Verifica se houve erro ao aceitar ou se o novo PCB é nulo
    if (err != ERR_OK || novo_pcb_cliente == NULL) {
        if (novo_pcb_cliente) fechar_conexao_cliente(novo_pcb_cliente); // Fecha
o novo PCB se ele foi criado
        return ERR_VAL; // Retorna erro de valor inválido
    }
```

```
    // Este servidor simples lida com apenas um cliente por vez.
    // Se já houver um cliente conectado (g_pcb_cliente não é NULL), recusa a
nova conexão.
    if (g_pcb_cliente != NULL) {
        fechar_conexao_cliente(novo_pcb_cliente); // Fecha a nova conexão
        return ERR_ABRT; // Indica ao LWIP para abortar esta conexão
    }
```

```
    g_pcb_cliente = novo_pcb_cliente; // Armazena o PCB do novo cliente
conectado
```

```
    // Configura os callbacks para a nova conexão do cliente
    tcp_setprio(g_pcb_cliente, TCP_PRIO_NORMAL); // Define a prioridade da
conexão
    tcp_arg(g_pcb_cliente, NULL); // Define um argumento para ser passado aos
callbacks (não usado aqui)
    tcp_recv(g_pcb_cliente, server_recv_cb); // Define o callback para quando
dados são recebidos
    tcp_err(g_pcb_cliente, server_err_cb); // Define o callback para quando
erros ocorrem
    return ERR_OK; // Conexão aceita com sucesso
}
```

```
/**
```

\* Inicializa o servidor TCP.

\* Cria um PCB, faz o bind para a porta e endereço, e começa a escutar por conexões.

\* true Se o servidor foi inicializado com sucesso.

\* false Se ocorreu algum erro durante a inicialização.

\*/

```
bool init_servidor_tcp(void) {
```

```
    // Cria um novo PCB (Protocol Control Block) para escutar por conexões TCP
```

```
    g_pcb_escuta = tcp_new_ip_type(IPADDR_TYPE_ANY); //  
    IPADDR_TYPE_ANY para escutar em qualquer interface de rede
```

```
    if (!g_pcb_escuta) {
```

```
        pisca_led(PINO_LED_ERRO, 5, 200); // 5 piscadas = erro ao criar PCB  
        return false;
```

```
    }
```

```
    // Associa (bind) o PCB a qualquer endereço IP local e à porta TCP definida
```

```
    err_t erro_bind = tcp_bind(g_pcb_escuta, IP_ANY_TYPE, PORTA_TCP);
```

```
    if (erro_bind != ERR_OK) {
```

```
        fechar_conexao_cliente(g_pcb_escuta); g_pcb_escuta = NULL; // Limpa o  
        PCB de escuta
```

```
        pisca_led(PINO_LED_ERRO, 6, 200); // 6 piscadas = erro no bind  
        return false;
```

```
    }
```

```
    // Coloca o PCB no estado de escuta (LISTEN), pronto para aceitar conexões
```

```
    // O backlog de 1 significa que apenas 1 conexão pode ficar na fila se o  
    servidor estiver ocupado.
```

```
    struct tcp_pcb *pcb_temporario_escuta =  
    tcp_listen_with_backlog(g_pcb_escuta, 1);
```

```
    if (!pcb_temporario_escuta) { // Se tcp_listen falhar, o PCB original é liberado  
    por LWIP
```



```

        if (g_pcb_escuta) fechar_conexao_cliente(g_pcb_escuta); // Segurança
extra
        g_pcb_escuta = NULL;
        pisca_led(PINO_LED_ERRO, 7, 200); // 7 piscadas = erro ao escutar
        return false;
    }

    g_pcb_escuta = pcb_temporario_escuta; // Atualiza para o PCB retornado
por tcp_listen

    // Define a função de callback (server_accept_cb) que será chamada quando
uma nova conexão for aceita
    tcp_accept(g_pcb_escuta, server_accept_cb);
    return true; // Servidor inicializado com sucesso
}

// --- Função Principal ---
int main() {

    // Inicializa os pinos GPIO para os LEDs
    gpio_init(PINO_LED_ERRO); gpio_set_dir(PINO_LED_ERRO, GPIO_OUT);
    gpio_init(PINO_LED_OK);  gpio_set_dir(PINO_LED_OK, GPIO_OUT);

    // Inicializa o pino GPIO para o botão
    gpio_init(PINO_BOTAO);
    gpio_set_dir(PINO_BOTAO, GPIO_IN); // Configura como entrada
    gpio_pull_up(PINO_BOTAO);        // Habilita resistor de pull-up interno

    // Inicializa o pino GPIO para o sensor DHT11
    gpio_init(PINO_DHT11); // A direção (entrada/saída) será gerenciada pela
função ler_dht11()

    // Inicializa o chip Wi-Fi CYW43

```

```

if (cyw43_arch_init()) {
    // Erro crítico: não conseguiu inicializar o hardware Wi-Fi
    while (true) pisca_led(PINO_LED_ERRO, 1, 500); // Pisca LED de erro
    continuamente
}

cyw43_arch_enable_sta_mode(); // Habilita o modo "Station" (para conectar
a um roteador Wi-Fi)

// Tenta conectar à rede Wi-Fi especificada
if (cyw43_arch_wifi_connect_timeout_ms(WIFI_SSID, WIFI_PASS,
CYW43_AUTH_WPA2_AES_PSK, TIMEOUT_CONEXAO_WIFI_MS)) {
    // Erro: não conseguiu conectar ao Wi-Fi
    while (true) pisca_led(PINO_LED_ERRO, 2, 700); // Pisca LED de erro
    continuamente
}

// Se chegou aqui, o Wi-Fi está conectado
gpio_put(PINO_LED_OK, 1); // Acende o LED de OK para indicar que o Wi-
Fi está conectado e o sistema pronto

// Inicializa o servidor TCP
if (!init_servidor_tcp()) {
    // Erro: não conseguiu inicializar o servidor TCP
    // A função init_servidor_tcp() já terá acionado o LED de erro com um
    padrão específico.
    while(true) { // Mantém o Pico funcionando para que o LED de erro
    continue piscando
        cyw43_arch_poll(); // Continua processando eventos de rede
        sleep_ms(100);
    }
}

// Se chegou aqui, o servidor web está pronto e escutando por conexões
// Loop principal do programa
while (true) {

```

```
    cyw43_arch_poll(); // ESSENCIAL: Processa todos os eventos pendentes  
da rede Wi-Fi e da pilha TCP/IP LwIP
```

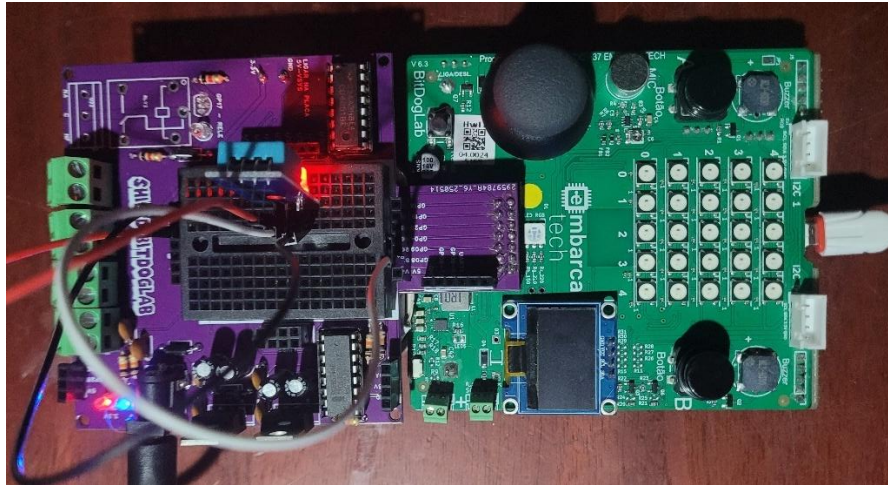
```
    sleep_ms(10);    // Pequena pausa para não sobrecarregar a CPU, mas  
mantém a responsividade
```

```
}
```

```
return 0; // Esta linha nunca é alcançada em um sistema embarcado típico
```

```
}
```

## Resultados obtidos



BitDogLab conectada com o DHT11



BitDogLab rodando como servidor Web e exibindo os valores

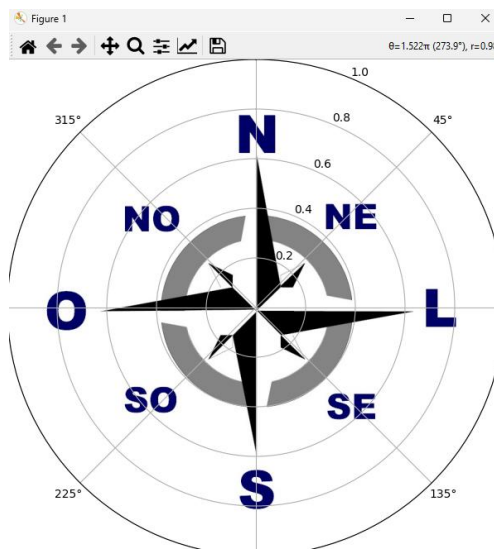
Este código transforma a placa BitDogLab em um servidor Web para monitoramento de sensores e do botão.

Essencialmente, o programa primeiro se conecta a uma rede Wi-Fi usando as credenciais definidas. Após a conexão, ele inicializa um servidor TCP que fica aguardando por requisições na porta 8081.

Quando você acessa o endereço IP da placa em um navegador, a BitDogLab recebe sua requisição, lê instantaneamente os dados de um sensor de temperatura e umidade (DHT11) e o estado de um botão. Em seguida, ele insere esses valores em um template HTML, que já possui um estilo visual definido, e envia essa página completa de volta ao seu navegador. A página é configurada para se recarregar a cada segundo, exibindo os dados do sensor em tempo real. O código também utiliza LEDs externos para sinalizar visualmente o status da conexão e possíveis erros.

## Enunciado 2:

Leitura da posição do joystick da placa BitDogLab, para que seja visualizado em um servidor com base no código apresentado na aula do capítulo 3, da unidade 2, crie um programa para ler a posição do joystick e enviar a posição X e Y para um servidor via Wi-Fi. Além disso, como um desafio extra, crie uma rosa dos ventos imaginária e envie para o aplicativo a posição (Norte, Sul, Leste, Oeste, Nordeste, Sudeste, Noroeste e Sudoeste) selecionada no joystick.



Tela Aplicativo server.py

## Código do server.py

```
import socket
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

# Configurações do socket UDP para receber dados do joystick
IP_UDP = "0.0.0.0"    # Escuta em todas as interfaces de rede
PORTA_UDP = 8081      # Porta para escutar os dados UDP

# Criação do socket UDP
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind((IP_UDP, PORTA_UDP)) # Vincula socket ao IP e porta
sock.setblocking(False)       # Configura socket para modo não bloqueante

def vr_x_vr_y_para_direcao(vrx, vry):
    """
    Converte valores brutos VRX e VRY do joystick em direção da rosa dos ventos.

    Parâmetros:
        vr_x (int): valor do eixo X do joystick (0 a 4095)
        vr_y (int): valor do eixo Y do joystick (0 a 4095)

    Retorna:
        (str, float): direção (ex: "N", "NE", etc.) e ângulo em radianos,
                     ajustado para 0 rad ser Norte e sentido horário
    """
    # Normaliza VRX e VRY para intervalo -1 a 1, com 0 no centro do joystick
```

```

x = (vrx - 2048) / 2048.0
y = (vry - 2048) / 2048.0

# Calcula ângulo do vetor (x,y) usando arctan2
angulo = np.arctan2(y, x)

# Ajusta ângulo para que 0 rad seja no Norte e ângulo cresça no sentido
horário
angulo = (np.pi/2 - angulo) % (2*np.pi)

# Define setores da rosa dos ventos (8 direções principais)
setores = ["N", "NE", "E", "SE", "S", "SO", "O", "NO"] # Norte, Nordeste,
Leste, Sudeste, Sul, Sudoeste, Oeste, Noroeste

# Calcula índice do setor a partir do ângulo
indice_setor = int((angulo + np.pi/8) // (np.pi/4)) % 8

return setores[indice_setor], angulo

# Cria a figura principal do matplotlib para exibir a rosa dos ventos e a seta
figura = plt.figure(figsize=(6,6))

# 1) Eixo cartesiano para a imagem de fundo (rosa dos ventos)
eixo_imagem = figura.add_axes([0, 0, 1, 1], zorder=0)
imagem_fundo = mpimg.imread('ROSA DOS VENTOS.jpg') # Carrega imagem
quadrada da rosa dos ventos
eixo_imagem.imshow(imagem_fundo)
eixo_imagem.axis('off') # Remove os eixos para mostrar só a imagem

# 2) Eixo polar transparente para desenhar a seta da direção do joystick sobre
a imagem
eixo_polar = figura.add_axes([0, 0, 1, 1], polar=True, zorder=1)

```

```

eixo_polar.set_theta_zero_location('N') # Define zero graus apontando para
cima (Norte)

eixo_polar.set_theta_direction(-1)     # Faz ângulo crescer no sentido horário

eixo_polar.set_rlim(0, 1)             # Limita raio do gráfico polar entre 0 e 1


# Torna fundo do eixo polar transparente para imagem ficar visível
eixo_polar.patch.set_alpha(0)


# Define as direções fixas para anotação da rosa dos ventos (em graus e
rótulos)
graus_direcoes = np.arange(0, 360, 45)

rotulos_direcoes = ["N", "NE", "E", "SE", "S", "SO", "O", "NO"] # Norte,
Nordeste, Leste, Sudeste, Sul, Sudoeste, Oeste, Noroeste


# Coloca as anotações das direções em torno do círculo, levemente fora do
raio 1
for grau, rotulo in zip(graus_direcoes, rotulos_direcoes):
    eixo_polar.annotate(rotulo,
                        (np.deg2rad(grau), 1.05), # posição em radianos e raio
                        levemente maior que 1
                        ha='center',             # alinha texto horizontalmente no centro
                        va='center',             # alinha texto verticalmente no centro
                        fontsize=12,
                        fontweight='bold')


# Cria uma linha (seta) que indicará a direção do joystick
linha_set, = eixo_polar.plot([], [], color='r', lw=3, marker='>', markersize=10)


def interpretar_mensagem(mensagem):
    """
    Interpreta mensagem recebida via UDP e extrai valores VRX e VRY.

```



Parâmetros:

mensagem (str): string no formato 'VRX=xxxx VRY=xxxx'

Retorna:

(int, int): valores inteiros de VRX e VRY; retorna (None, None) se falhar

"""

try:

partes = mensagem.strip().split()

vrx = int(partes[0].split('=')[1])

vry = int(partes[1].split('=')[1])

return vrx, vry

except:

return None, None

def atualizar\_sete(angulo):

"""

Atualiza a posição da seta no gráfico polar conforme o ângulo recebido.

Parâmetros:

angulo (float): ângulo em radianos para onde a seta deve apontar

"""

raio = [0, 0.9]            # Seta vai do centro (0) até 90% do raio

theta = [angulo, angulo] # Mesma direção para início e fim da linha

linha\_sete.set\_data(theta, raio)

print(f"Escutando dados UDP em {IP\_UDP}:{PORTA\_UDP}...")

def loop\_principal():

"""

Loop principal que fica escutando mensagens UDP, processa dados do joystick,

atualiza o gráfico e controla a exibição da seta conforme o movimento.

"""

while True:

try:

# Tenta receber dados do socket UDP

dados, endereco = sock.recvfrom(1024)

mensagem = dados.decode()

vrx, vry = interpretar\_mensagem(mensagem)

# Ignora se não conseguiu interpretar dados

if vrx is None or vry is None:

continue

# Normaliza os valores VRX e VRY para intervalo [-1, 1]

x = (vrx - 2048) / 2048.0

y = (vry - 2048) / 2048.0

# Calcula magnitude do vetor joystick para detectar zona morta

zona\_morta = 0.1 # Ajuste para ignorar pequenas oscilações

magnitude = (x\*\*2 + y\*\*2)\*\*0.5

if magnitude < zona\_morta:

# Joystick parado, esconde a seta do gráfico

linha\_sete.set\_visible(False)

else:

# Joystick em movimento, mostra seta e atualiza direção

linha\_sete.set\_visible(True)

direcao, angulo = vrx\_vry\_para\_direcao(vrx, vry)

print(f"Direção joystick: {direcao} (ângulo {np.rad2deg(angulo):.1f}°)

VRX={vrx} VRY={vry}")

atualizar\_sete(angulo)

```
plt.pause(0.01) # Atualiza gráfico com pequeno delay para não travar interface
```

```
except BlockingIOError:
```

```
    # Nenhum dado recebido no momento, apenas pausa para não travar o loop
```

```
    plt.pause(0.01)
```

```
except Exception as e:
```

```
    print(f"Erro inesperado: {e}")
```

```
    plt.pause(0.01)
```

```
if __name__ == "__main__":
```

```
    plt.ion() # Ativa modo interativo do matplotlib para atualização em tempo real
```

```
    plt.show() # Exibe a janela do gráfico
```

```
    loop_principal()
```

## Código C Bare Metal

```
#include <stdio.h>
#include <string.h>
#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"
#include "hardware/adc.h"
#include "lwip/pbuf.h"
#include "lwip/udp.h"
#include "lwip/ip_addr.h"

// ==== CONFIGURAÇÕES ====
#define WIFI_SSID "copelli4" //Nome da rede
#define WIFI_PASSWORD "copelli4" //Senha da rede
#define NOTEBOOK_IP "192.168.0.228" // IP do notebook
#define UDP_PORT 8081

// ==== LEDS ====
#define LED_WIFI_OK 11
#define LED_WIFI_ERR 12
#define LED_STATUS 13

// ==== JOYSTICK ====
#define JOY_VRX 27 // ADC1
#define JOY_VRY 26 // ADC0
#define JOY_SW 22 // Digital

// ==== VARIÁVEIS ====
struct udp_pcb *udp_conn;
```

```
ip_addr_t notebook_addr;
```

```
void init_leds() {  
    gpio_init(LED_WIFI_OK);  
    gpio_init(LED_WIFI_ERR);  
    gpio_init(LED_STATUS);  
    gpio_set_dir(LED_WIFI_OK, GPIO_OUT);  
    gpio_set_dir(LED_WIFI_ERR, GPIO_OUT);  
    gpio_set_dir(LED_STATUS, GPIO_OUT);  
    gpio_put(LED_WIFI_OK, 0);  
    gpio_put(LED_WIFI_ERR, 0);  
    gpio_put(LED_STATUS, 0);  
}
```

```
void init_joystick() {  
    adc_init();  
    adc_gpio_init(JOY_VRX);  
    adc_gpio_init(JOY_VRY);  
    gpio_init(JOY_SW);  
    gpio_set_dir(JOY_SW, GPIO_IN);  
    gpio_pull_up(JOY_SW);  
  
}
```

```
uint16_t read_adc(uint channel) {  
    adc_select_input(channel);  
    return adc_read();  
}
```

```
bool connect_wifi() {
```

```

printf("Conectando ao Wi-Fi...\n");
if (cyw43_arch_init()) {
    printf("Erro ao inicializar Wi-Fi\n");
    gpio_put(LED_WIFI_ERR, 1);
    return false;
}

cyw43_arch_enable_sta_mode();

if (cyw43_arch_wifi_connect_timeout_ms(WIFI_SSID, WIFI_PASSWORD,
CYW43_AUTH_WPA2_AES_PSK, 20000)) {
    printf("Falha na conexão Wi-Fi\n");
    gpio_put(LED_WIFI_ERR, 1);
    return false;
}

printf("Conectado! IP: %s\n", ip4addr_ntoa(netif_ip4_addr(netif_default)));
gpio_put(LED_WIFI_OK, 1);
return true;
}

bool setup_udp() {
    if (!ipaddr_aton(NOTEBOOK_IP, &notebook_addr)) {
        printf("IP do notebook inválido\n");
        gpio_put(LED_WIFI_ERR, 1);
        return false;
    }

    udp_conn = udp_new();
    if (!udp_conn) {
        printf("Erro ao criar socket UDP\n");
    }
}

```

```

        gpio_put(LED_WIFI_ERR, 1);
        return false;
    }

    printf("Configurado para enviar para %s:%d\n",
ip4addr_ntoa(&notebook_addr), UDP_PORT);
    return true;
}

void send_udp_message(const char *message) {
    struct pbuf *p = pbuf_alloc(PBUF_TRANSPORT, strlen(message),
PBUF_RAM);
    if (!p) {
        printf("Erro alocando buffer\n");
        return;
    }

    memcpy(p->payload, message, strlen(message));
    err_t err = udp_sendto(udp_conn, p, &notebook_addr, UDP_PORT);
    pbuf_free(p);

    if (err != ERR_OK) {
        printf("Erro enviando mensagem: %d\n", err);
        gpio_put(LED_STATUS, 0);
    } else {
        printf("Mensagem enviada: %s\n", message);
        gpio_put(LED_STATUS, 1);
    }
}

int main() {

```

```

stdio_init_all();
init_leds();
init_joystick();

if (!connect_wifi() || !setup_udp()) {
    while (1) sleep_ms(1000); // Loop de erro
}

while (true) {

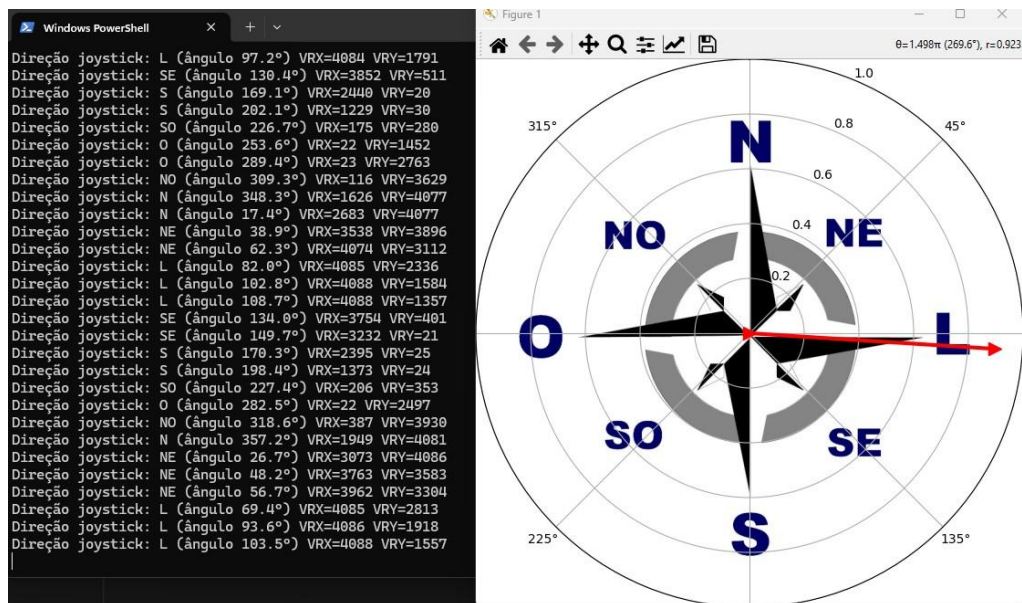
    bool joystick = !gpio_get(JOY_SW);
    // Leitura do joystick
    uint16_t x = read_adc(1);
    uint16_t y = read_adc(0);

    char msg[128];
    snprintf(msg, sizeof(msg),
        "VRX=%u VRY=%u",
        x, y
    );
    send_udp_message(msg);
    //step++;
    sleep_ms(100);
}
}

```



## Resultados obtidos:



Ao girar o joystick a seta acompanha mostrando a direção

O sistema funciona como uma solução de controle remoto sem fio, onde o código em C para a placa BitDogLab atua como um cliente UDP, enquanto o script server.py em Python age como um servidor UDP e uma interface de visualização.

No lado da placa BitDogLab, o código C utiliza a biblioteca cyw43\_arch para se conectar a uma rede Wi-Fi e a pilha de rede LwIP para comunicação. Ele configura os pinos do conversor analógico-digital (ADC) para ler continuamente os valores brutos (0-4095) dos eixos X e Y de um joystick. A cada 100 milissegundos, ele formata esses dados em uma string simples, como VRX=2048, VRY=2048, e envia essa mensagem como um datagrama UDP para o endereço IP e porta fixos do notebook. LEDs na placa fornecem feedback visual sobre o status da conexão Wi-Fi e do envio de dados.



No lado do servidor, o script Python cria um socket UDP não-bloqueante que escuta na porta 8081 em todas as interfaces de rede. Utilizando a biblioteca Matplotlib, ele gera uma interface gráfica que exibe uma imagem de rosa dos ventos como fundo. Quando um pacote UDP é recebido, o script decodifica a mensagem, extrai os valores de VRX e VRY, e utiliza a biblioteca Numpy para converter essas coordenadas cartesianas em um ângulo polar. Este ângulo é então usado para atualizar, em tempo real, a posição de uma seta vermelha sobreposta à rosa dos ventos, indicando visualmente a direção do joystick. Uma zona morta é implementada para que a seta desapareça quando o joystick está centralizado, evitando oscilações.

## Enunciado Desafio:

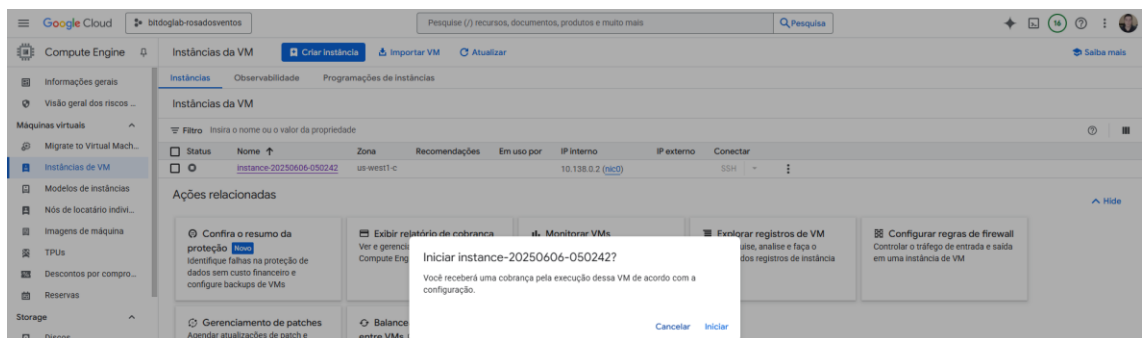
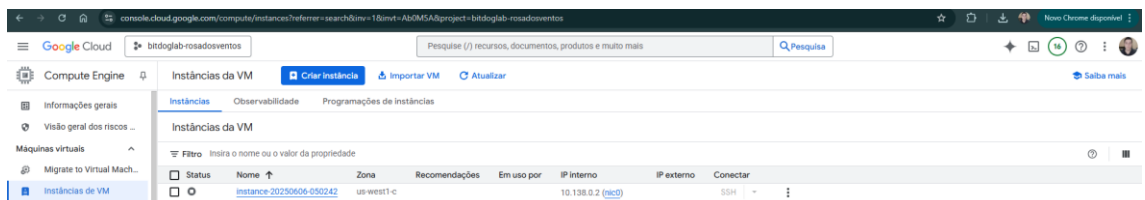
Servidor na nuvem: Refaça as tarefas anteriores, utilizando um servidor na nuvem, como por exemplo: AWS, Google e entre outros.

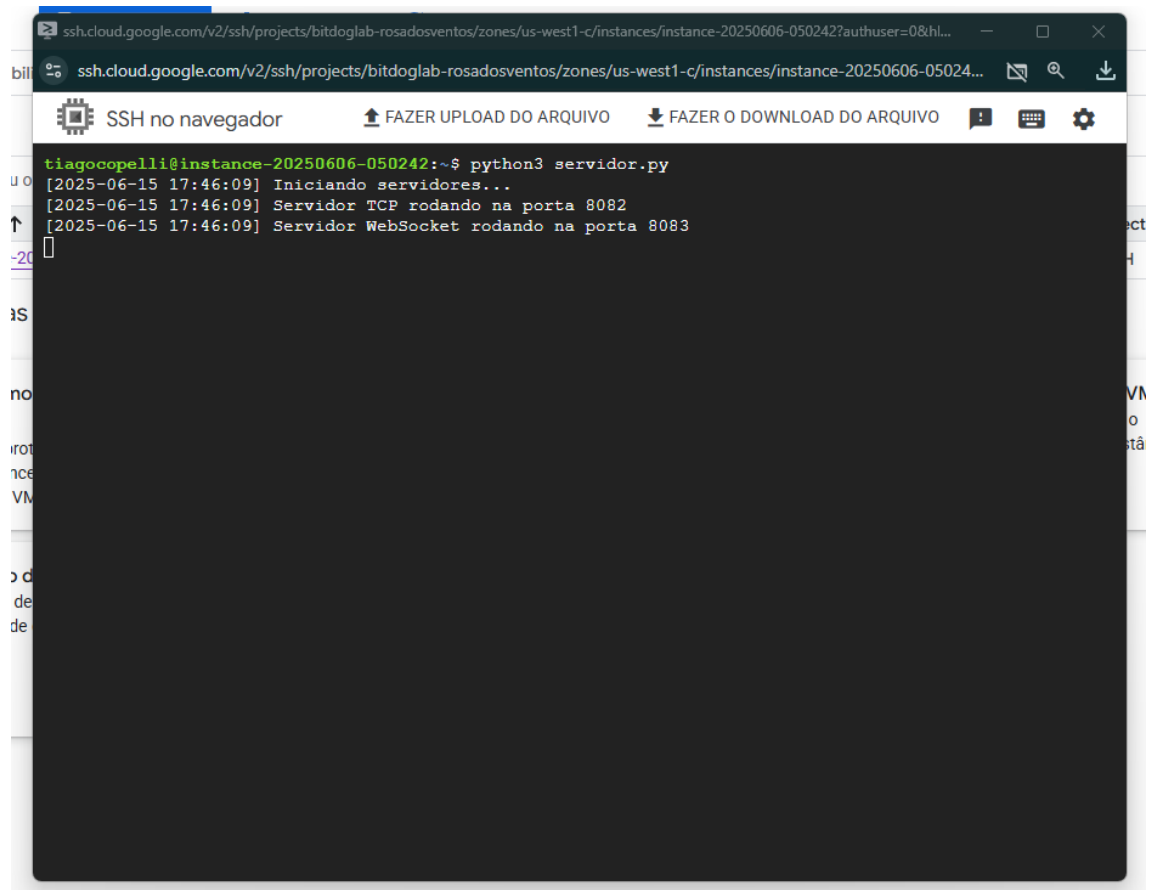
Utilizando o servidor do Google Cloud – Compute Engine

Você está trabalhando em [bitdoglab-rosadosventos](#)

Número do projeto: 831712877265  ID do projeto: bitdoglab-rosadosventos 

[Painel](#) [Cloud Hub](#) [Novo](#)





The image shows a web-based SSH terminal interface. The browser's address bar displays the URL: `ssh.cloud.google.com/v2/ssh/projects/bitdoglab-rosadosventos/zones/us-west1-c/instances/instance-20250606-050242?authuser=0&hl...`. The terminal window has a title bar with the text "SSH no navegador" and two buttons: "FAZER UPLOAD DO ARQUIVO" and "FAZER O DOWNLOAD DO ARQUIVO". The terminal output shows the following commands and messages:

```
tiagocopelli@instance-20250606-050242:~$ python3 servidor.py
[2025-06-15 17:46:09] Iniciando servidores...
[2025-06-15 17:46:09] Servidor TCP rodando na porta 8082
[2025-06-15 17:46:09] Servidor WebSocket rodando na porta 8083
```

The terminal is currently in a dark theme with a black background and white text. The prompt character is a green dollar sign.

## Código Dashboard.html

```
<!DOCTYPE html>

<html lang="pt-BR">

<head>

  <meta charset="UTF-8">

  <meta name="viewport" content="width=device-width, initial-
scale=1.0">

  <title>Dashboard do Joystick - Completo</title>

  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>

  <style>

    /* CSS RESET */

    * {

      margin: 0;

      padding: 0;

      box-sizing: border-box;

    }

    /* ESTILOS GERAIS */

    body {

      font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;

      background-color: #f5f5f5;

      color: #333;

      display: flex;

      flex-direction: column;

      align-items: center;

      justify-content: center;

      min-height: 100vh;

      padding: 20px;

    }
```

```
h1 {  
    color: #2c3e50;  
    margin-bottom: 20px;  
    text-align: center;  
    font-size: 2.2rem;  
    text-shadow: 1px 1px 2px rgba(0,0,0,0.1);  
}
```

```
/* CONTAINER PRINCIPAL */  
.dashboard-container {  
    background-color: white;  
    border-radius: 15px;  
    box-shadow: 0 10px 30px rgba(0, 0, 0, 0.1);  
    padding: 30px;  
    width: 100%;  
    max-width: 500px;  
    margin: 0 auto;  
    display: flex;  
    flex-direction: column;  
    align-items: center;  
}
```

```
/* ROSA DOS VENTOS */  
.compass-wrapper {  
    position: relative;  
    width: 100%;  
    max-width: 350px;  
    aspect-ratio: 1/1;  
    margin: 0 auto 30px;  
}
```

```
#compass {  
    width: 100%;  
    height: 100%;  
}
```

```
#arrow {  
    position: absolute;  
    width: 6px;  
    height: 120px;  
    background: linear-gradient(to bottom, #ff0000, #ff6b6b);  
    left: 50%;  
    top: 50%;  
    transform-origin: 50% 0;  
    z-index: 10;  
    border-radius: 3px;  
    box-shadow: 0 0 15px rgba(255, 0, 0, 0.5);  
    transition: transform 0.2s ease-out, height 0.2s ease-out;  
}
```

```
.compass-point {  
    position: absolute;  
    font-weight: bold;  
    font-size: 18px;  
    transform: translate(-50%, -50%);  
    text-shadow: 0 0 5px white;  
    z-index: 5;  
    user-select: none;  
}
```

```
/* DISPLAY DE INFORMAÇÕES */
```

```
.data-display {  
    display: grid;  
    grid-template-columns: 1fr 1fr;  
    gap: 15px;  
    width: 100%;  
    margin-top: 20px;  
}
```

```
.data-card {  
    background-color: #f8f9fa;  
    border-radius: 10px;  
    padding: 15px;  
    text-align: center;  
    box-shadow: 0 3px 10px rgba(0,0,0,0.05);  
}
```

```
.data-card h3 {  
    color: #7f8c8d;  
    font-size: 0.9rem;  
    margin-bottom: 5px;  
}
```

```
.data-card p {  
    font-size: 1.5rem;  
    font-weight: bold;  
    color: #2c3e50;  
}
```

```
/* STATUS DO BOTÃO CENTRAL */
```

```
.button-status {  
    margin-top: 20px;  
    padding: 15px;  
    border-radius: 10px;  
    text-align: center;  
    font-weight: bold;  
    font-size: 1.2rem;  
    transition: all 0.3s ease;  
    width: 100%;  
}
```

```
.button-pressed {  
    background-color: #ff4444;  
    color: white;  
    box-shadow: 0 0 15px rgba(255, 68, 68, 0.5);  
}
```

```
.button-released {  
    background-color: #4CAF50;  
    color: white;  
    box-shadow: 0 0 15px rgba(76, 175, 80, 0.5);  
}
```

```
/* NOVO: Estilo para os botões A e B */
```

```
.extra-buttons-container {  
    display: flex;  
    justify-content: center;  
    gap: 30px;
```



```
margin-top: 25px;  
width: 100%;  
}
```

```
.extra-button {  
  width: 80px;  
  height: 80px;  
  border-radius: 50%;  
  background-color: #4CAF50; /* Verde inicial */  
  color: white;  
  display: flex;  
  justify-content: center;  
  align-items: center;  
  font-weight: bold;  
  font-size: 1.8rem;  
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.15);  
  transition: background-color 0.2s ease-in-out;  
  border: 3px solid white;  
}
```

```
.extra-button.pressed {  
  background-color: #e74c3c; /* Vermelho quando pressionado */  
}
```

```
/* ANIMAÇÕES */  
@keyframes pulse {  
  0% { opacity: 0.8; }  
  50% { opacity: 1; }  
  100% { opacity: 0.8; }  
}
```

```

        .pulse {
            animation: pulse 1.5s infinite ease-in-out;
        }
    </style>
</head>
<body>
    <div class="dashboard-container">
        <h1>Dashboard do Joystick</h1>

        <div class="compass-wrapper">
            <canvas id="compass"></canvas>
            <div id="arrow"></div>

            <!-- Pontos cardeais e colaterais -->
            <div class="compass-point" style="top: 3%; left: 50%; color:
#e74c3c;">N</div>
            <div class="compass-point" style="top: 25%; right: 25%; color:
#e67e22;">NE</div>
            <div class="compass-point" style="top: 50%; right: 3%; color:
#3498db;">L</div>
            <div class="compass-point" style="bottom: 25%; right: 25%;
color: #1abc9c;">SE</div>
            <div class="compass-point" style="bottom: 3%; left: 50%; color:
#e74c3c;">S</div>
            <div class="compass-point" style="bottom: 25%; left: 25%; color:
#9b59b6;">SO</div>
            <div class="compass-point" style="top: 50%; left: 3%; color:
#3498db;">O</div>
            <div class="compass-point" style="top: 25%; left: 25%; color:
#f1c40f;">NO</div>
        </div>
    </div>

```

```
<div class="data-display">
  <div class="data-card">
    <h3>Direção</h3>
    <p id="angle-display">0°</p>
  </div>
  <div class="data-card">
    <h3>Intensidade</h3>
    <p id="intensity-display">0%</p>
  </div>
```

```
<div class="data-card">
  <h3>Temperatura</h3>
  <p id="temp-display">-- °C</p>
</div>
```

```
<div class="data-card">
  <h3>Umidade</h3>
  <p id="humi-display">-- %</p>
</div>
</div>
```

```
<div id="button-status" class="button-status button-released">
  BOTÃO JOYSTICK: SOLTO
</div>
```

```
<!-- NOVO: Container para os botões A e B -->
<div class="extra-buttons-container">
  <div id="button-a" class="extra-button">A</div>
```

```
        <div id="button-b" class="extra-button">B</div>
    </div>
</div>
```

```
<script>
    // --- ELEMENTOS DA INTERFACE ---

    const arrow = document.getElementById('arrow');
    const angleDisplay = document.getElementById('angle-display');
    const intensityDisplay = document.getElementById('intensity-
display');
    const buttonStatus = document.getElementById('button-status');
    const buttonA = document.getElementById('button-a');
    const buttonB = document.getElementById('button-b');
    const h1 = document.querySelector('h1');

    const tempDisplay = document.getElementById('temp-display');
    const humiDisplay = document.getElementById('humi-display');

    // Cria o gráfico da rosa dos ventos (código original, sem alterações)
    const ctx = document.getElementById('compass').getContext('2d');

    const compassChart = new Chart(ctx, { type: 'doughnut', data: {
labels: ['N', 'NE', 'E', 'SE', 'S', 'SO', 'O', 'NO'], datasets: [{ data: [1, 1, 1, 1, 1, 1, 1, 1], backgroundColor: ['#e74c3c20', '#e67e2220', '#3498db20', '#1abc9c20', '#e74c3c20', '#9b59b620', '#3498db20', '#f1c40f20'], borderColor: ['#e74c3c', '#e67e22', '#3498db', '#1abc9c', '#e74c3c', '#9b59b6', '#3498db', '#f1c40f'], borderWidth: 1 } ] }, options: { cutout: '75%', rotation: -45, plugins: { legend: { display: false }, tooltip: { enabled: false } }, animation: { animateRotate: false }, responsive: true, maintainAspectRatio: false } });

    // --- FUNÇÕES DE ATUALIZAÇÃO DA UI ---

    function updateArrowPosition(angle, intensity) {
        const degrees = ((angle * 180 / Math.PI) % 360).toFixed(1);
```

```

    const percent = (intensity * 100).toFixed(0);
    arrow.style.transform = `translateX(-50%) rotate(${angle}rad)`;
    arrow.style.height = `${80 + (intensity * 70)}px`;
    angleDisplay.textContent = `${degrees}°`;
    intensityDisplay.textContent = `${percent}%`;
    if (intensity > 0.1) { arrow.classList.add('pulse'); } else {
arrow.classList.remove('pulse'); }
}

```

```

function updateButtonStatus(element, isPressed, textPrefix) {
    if (isPressed) {
        element.textContent = `${textPrefix}: PRESSIONADO`;
        element.classList.remove('button-released');
        element.classList.add('button-pressed');
    } else {
        element.textContent = `${textPrefix}: SOLTO`;
        element.classList.remove('button-pressed');
        element.classList.add('button-released');
    }
}

```

```

function updateExtraButtonStatus(element, isPressed) {
    if (isPressed) {
        element.classList.add('pressed');
    } else {
        element.classList.remove('pressed');
    }
}

```

```

function processJoystickData(vrx, vry) {
    const ADC_CENTER = 2048;

```

```

const ADC_MAX_DEV = 2048;
const x_norm = (vrx - ADC_CENTER) / ADC_MAX_DEV;
const y_norm = -((vry - ADC_CENTER) / ADC_MAX_DEV);
const angle = Math.atan2(y_norm, x_norm) + Math.PI / 2;
const intensity = Math.min(Math.sqrt(x_norm * x_norm + y_norm
* y_norm), 1.0);
return { angle, intensity };
}

```

// --- LÓGICA DO WEBSOCKET ---

```

function connectWebSocket() {
    // Use o IP público do seu servidor Google Cloud
    const socket = new WebSocket("ws://34.127.94.4:8083");

    socket.onopen = function(e) {
        h1.textContent = "Joystick Conectado";
    };

    socket.onmessage = function(event) {
        try {
            const data = JSON.parse(event.data);

            if (data.status) {
                console.log("Status recebido:", data.status);
                return;
            }

            const { angle, intensity } = processJoystickData(data.VRX,
data.VRY);

            updateArrowPosition(angle, intensity);

```

```

        updateButtonStatus(buttonStatus, (data.BTN === '1'),
'JOYSTICK');

        updateExtraButtonStatus(buttonA, (data.A === '1'));
        updateExtraButtonStatus(buttonB, (data.B === '1'));

        if (data.TEMP !== undefined) {
            tempDisplay.textContent =
`$${parseFloat(data.TEMP).toFixed(1)} °C`;
        }

        if (!isNaN(data.UMI)) {
            humiDisplay.textContent =
`$${parseFloat(data.UMI).toFixed(1)} %`;
        } else
        {
            humiDisplay.textContent = '-- %';
        }

    } catch (error) {
        console.error("Erro ao processar mensagem:", error);
    }
};

socket.onclose = function(event) {
    h1.textContent = "Conexão Perdida...";
    setTimeout(connectWebSocket, 3000); // Tenta reconectar
};

socket.onerror = function(error) {
    h1.textContent = "Erro de Conexão";
};
}

```

```
// Inicia a conexão e o estado inicial da UI
connectWebSocket();
updateArrowPosition(0, 0);
updateButtonStatus(buttonStatus, false, 'JOYSTICK');
updateExtraButtonStatus(buttonA, false);
updateExtraButtonStatus(buttonB, false);
</script>
</body>
</html>
```



## Código C Bare Metal

```
#include <stdio.h>

#include <string.h>

#include <stdbool.h>


#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"
#include "hardware/adc.h"
#include "hardware/gpio.h"
#include "pico/time.h"


#include "lwip/pbuf.h"
#include "lwip/tcp.h"
#include "lwip/ip_addr.h"


//
=====

// ==== CONFIGURAÇÕES GERAIS ====

//
=====

#define WIFI_SSID "copelli4"
#define WIFI_PASSWORD "copelli4"
#define IP_SERVIDOR "34.127.94.4" // IP do seu servidor na nuvem
#define PORTA_TCP 8082


//
=====

// ==== DEFINIÇÃO DE PINOS ====
```

```

//
=====
=====

// LEDs de estado
#define LED_WIFI_CONECTADO 11
#define LED_WIFI_ERRO 12
#define LED_ESTADO 13


// Periféricos de entrada
#define PINO_JOY_VRX 27    // Eixo X do Joystick (ADC 1)
#define PINO_JOY_VRY 26    // Eixo Y do Joystick (ADC 0)
#define PINO_JOY_BOTAO 22  // Botão do Joystick
#define PINO_BOTAO_A 5     // Botão extra 'A'
#define PINO_BOTAO_B 6     // Botão extra 'B'
#define PINO_DHT 16        // Pino de dados do sensor DHT11


//
=====
=====

// ==== BIBLIOTECA DO SENSOR DHT11 (Integrada) ====

//
=====
=====


#define TIMEOUT_DHT 200


// Estrutura para guardar os resultados da leitura do DHT
typedef struct {
    float umidade;
    float temperatura_c;
    bool valido; // Flag para indicar se a leitura foi bem-sucedida
} resultado_dht_t;

```

```
// Função interna para esperar uma mudança no estado do pino
static int aguardar_nivel_pino(uint pino_gpio, bool nivel, uint timeout_us) {
    uint contador = 0;
    while (gpio_get(pino_gpio) != nivel) {
        if (contador++ > timeout_us) return 0; // Timeout
        sleep_us(1);
    }
    return contador;
}
```

```
// Função para inicializar o pino do DHT
static void dht_inicializar(uint pino_gpio) {
    gpio_init(pino_gpio);
}
```

```
// Função para ler os dados do sensor (bloqueante)
static resultado_dht_t dht_ler_bloqueante(uint pino_gpio) {
    uint8_t dados[5] = {0, 0, 0, 0, 0};
    resultado_dht_t resultado = {0.0f, 0.0f, false};
```

```
    // 1. Sinal de início enviado pelo Pico
    gpio_set_dir(pino_gpio, GPIO_OUT);
    gpio_put(pino_gpio, 0);
    sleep_ms(20); // Espera pelo menos 18ms
    gpio_put(pino_gpio, 1);
    sleep_us(40);
    gpio_set_dir(pino_gpio, GPIO_IN);
```

```
    // 2. Espera pela resposta do DHT
    if (!aguardar_nivel_pino(pino_gpio, 0, TIMEOUT_DHT)) return resultado;
```

```

if (!aguardar_nivel_pino(pino_gpio, 1, TIMEOUT_DHT)) return resultado;
if (!aguardar_nivel_pino(pino_gpio, 0, TIMEOUT_DHT)) return resultado;

// 3. Leitura dos 40 bits de dados
for (int i = 0; i < 40; i++) {
    aguardar_nivel_pino(pino_gpio, 1, TIMEOUT_DHT);
    uint contagem_baixo = aguardar_nivel_pino(pino_gpio, 0,
TIMEOUT_DHT);
    if (contagem_baixo > 50) { // Se o pulso em nível alto for longo, é bit '1'
        dados[i / 8] |= (1 << (7 - (i % 8)));
    }
}

// 4. Verificação do checksum
if (((dados[0] + dados[1] + dados[2] + dados[3]) & 0xFF) != dados[4]) {
    printf("Erro de checksum do DHT\n");
    return resultado; // Checksum inválido
}

// 5. Processamento dos dados
resultado.umidade = (float)dados[0] + (float)dados[1] / 10.0f;
resultado.temperatura_c = (float)dados[2] + (float)dados[3] / 10.0f;
resultado.valido = true;

return resultado;
}

//
=====
=====

// ==== LÓGICA DE CONEXÃO TCP (LWIP) ====

```

```
//
=====

// Estrutura para manter o estado da conexão TCP
typedef struct CLIENTE_TCP_T_ {
    struct tcp_pcb *pcb_tcp;
    ip_addr_t endereco_remoto;
    bool conectado;
} cliente_tcp_t;

// Protótipos das funções de callback TCP
err_t callback_cliente_tcp_conectado(void *arg, struct tcp_pcb *tpcb, err_t erro);
void callback_cliente_tcp_erro(void *arg, err_t erro);
err_t callback_cliente_tcp_enviado(void *arg, struct tcp_pcb *tpcb, u16_t
tamanho);
void cliente_tcp_fechar_conexao(cliente_tcp_t *estado);

// Função para enviar os dados via TCP
void cliente_tcp_enviar_dados(cliente_tcp_t *estado, const char *mensagem) {
    if (!estado->conectado || estado->pcb_tcp == NULL) {
        printf("Não conectado. Impossível enviar dados.\n");
        return;
    }

    printf("Enviando: %s", mensagem); // a mensagem já tem \n
    err_t erro = tcp_write(estado->pcb_tcp, mensagem, strlen(mensagem),
TCP_WRITE_FLAG_COPY);

    if (erro != ERR_OK) {
        printf("Erro ao escrever para o buffer TCP: %d\n", erro);
        return;
    }
}
```

```

    }

    erro = tcp_output(estado->pcb_tcp);
    if (erro != ERR_OK) {
        printf("Erro ao enviar dados TCP: %d\n", erro);
    }
}

// Função para fechar a conexão TCP
void cliente_tcp_fechar_conexao(cliente_tcp_t *estado) {
    if (estado->pcb_tcp != NULL) {
        tcp_arg(estado->pcb_tcp, NULL);
        tcp_sent(estado->pcb_tcp, NULL);
        tcp_err(estado->pcb_tcp, NULL);
        tcp_close(estado->pcb_tcp);
        estado->pcb_tcp = NULL;
        estado->conectado = false;
        gpio_put(LED_ESTADO, 0);
        printf("Conexão TCP fechada.\n");
    }
}

// Callback de erro
void callback_cliente_tcp_erro(void *arg, err_t erro) {
    cliente_tcp_t *estado = (cliente_tcp_t*)arg;
    printf("Erro TCP: %d. Fechando conexão.\n", erro);
    cliente_tcp_fechar_conexao(estado);
}

// Callback de dados enviados

```

```
err_t callback_cliente_tcp_enviado(void *arg, struct tcp_pcb *pcb_tcp, u16_t tamanho) {
```

```
    gpio_put(LED_ESTADO, 1); // Pisca o LED para indicar envio
```

```
    sleep_ms(50);
```

```
    gpio_put(LED_ESTADO, 0);
```

```
    return ERR_OK;
```

```
}
```

```
// Callback de conexão estabelecida
```

```
err_t callback_cliente_tcp_conectado(void *arg, struct tcp_pcb *pcb_tcp, err_t erro) {
```

```
    cliente_tcp_t *estado = (cliente_tcp_t*)arg;
```

```
    if (erro != ERR_OK) {
```

```
        printf("Falha na conexão TCP: %d\n", erro);
```

```
        cliente_tcp_fechar_conexao(estado);
```

```
        return erro;
```

```
    }
```

```
    estado->conectado = true;
```

```
    printf("Conexão TCP estabelecida com sucesso!\n");
```

```
// Configura os outros callbacks
```

```
tcp_sent(pcb_tcp, callback_cliente_tcp_enviado);
```

```
// Envia uma mensagem inicial
```

```
cliente_tcp_enviar_dados(estado, "Olá do RP2040!\n");
```

```
return ERR_OK;
```

```
}
```

```
// Função para iniciar a conexão TCP
```

```
bool cliente_tcp_conectar(cliente_tcp_t *estado) {
```

```
    printf("Iniciando conexão com %s:%d\n", ip4addr_ntoa(&estado->endereco_remoto), PORTA_TCP);
```

```
    estado->pcb_tcp = tcp_new_ip_type(IP_GET_TYPE(&estado->endereco_remoto));
```

```
    if (estado->pcb_tcp == NULL) {  
        printf("Erro ao criar PCB.\n");  
        return false;  
    }
```

```
    tcp_arg(estado->pcb_tcp, estado);
```

```
    tcp_err(estado->pcb_tcp, callback_cliente_tcp_erro);
```

```
    err_t erro = tcp_connect(estado->pcb_tcp, &estado->endereco_remoto,  
PORTA_TCP, callback_cliente_tcp_conectado);
```

```
    return erro == ERR_OK;
```

```
}
```

```
//
```

```
=====
```

```
// ==== FUNÇÕES DE INICIALIZAÇÃO DE HARDWARE ====
```

```
//
```

```
=====
```

```
void inicializar_leds() {
```

```
    gpio_init(LED_WIFI_CONECTADO);
```

```
    gpio_init(LED_WIFI_ERRO);
```

```
    gpio_init(LED_ESTADO);
```

```
    gpio_set_dir(LED_WIFI_CONECTADO, GPIO_OUT);
```

```
    gpio_set_dir(LED_WIFI_ERRO, GPIO_OUT);
```

```
    gpio_set_dir(LED_ESTADO, GPIO_OUT);
```

```
    gpio_put(LED_WIFI_CONECTADO, 0);
```

```
    gpio_put(LED_WIFI_ERRO, 0);
```

```
    gpio_put(LED_ESTADO, 0);
```



```
}
```

```
void inicializar_periféricos() {
```

```
    // ADC para o Joystick
```

```
    adc_init();
```

```
    adc_gpio_init(PINO_JOY_VRX);
```

```
    adc_gpio_init(PINO_JOY_VRY);
```

```
    // Botões com pull-up interno
```

```
    gpio_init(PINO_JOY_BOTAO);
```

```
    gpio_set_dir(PINO_JOY_BOTAO, GPIO_IN);
```

```
    gpio_pull_up(PINO_JOY_BOTAO);
```

```
    gpio_init(PINO_BOTAO_A);
```

```
    gpio_set_dir(PINO_BOTAO_A, GPIO_IN);
```

```
    gpio_pull_up(PINO_BOTAO_A);
```

```
    gpio_init(PINO_BOTAO_B);
```

```
    gpio_set_dir(PINO_BOTAO_B, GPIO_IN);
```

```
    gpio_pull_up(PINO_BOTAO_B);
```

```
    // Sensor DHT
```

```
    dht_inicializar(PINO_DHT);
```

```
}
```

```
uint16_t ler_adc(uint canal) {
```

```
    adc_select_input(canal);
```

```
    return adc_read();
```

```
}
```

```

bool conectar_wifi() {
    printf("Conectando ao Wi-Fi...\n");
    if (cyw43_arch_init()) {
        printf("Erro ao inicializar Wi-Fi\n");
        gpio_put(LED_WIFI_ERRO, 1);
        return false;
    }
    cyw43_arch_enable_sta_mode();
    if (cyw43_arch_wifi_connect_timeout_ms(WIFI_SSID, WIFI_PASSWORD,
    CYW43_AUTH_WPA2_AES_PSK, 20000)) {
        printf("Falha na conexão Wi-Fi\n");
        gpio_put(LED_WIFI_ERRO, 1);
        return false;
    }
    printf("Conectado! IP: %s\n", ip4addr_ntoa(netif_ip4_addr(netif_default)));
    gpio_put(LED_WIFI_CONECTADO, 1);
    return true;
}

//
=====

// ==== FUNÇÃO PRINCIPAL (MAIN) ====

//
=====

int main() {
    stdio_init_all();

    inicializar_leds();
    inicializar_periféricos();

```

```
if (!conectar_wifi()) {  
    while (1) { tight_loop_contents(); } // Loop infinito em caso de falha no Wi-  
Fi  
}
```

```
// Prepara o estado do cliente TCP
```

```
cliente_tcp_t *estado_tcp = calloc(1, sizeof(cliente_tcp_t));
```

```
if (!estado_tcp) {
```

```
    printf("Erro ao alocar estado TCP\n");
```

```
    return 1;
```

```
}
```

```
ipaddr_aton(IP_SERVIDOR, &estado_tcp->endereco_remoto);
```

```
// Variáveis para guardar os dados dos sensores
```

```
float temperatura = 0.0f;
```

```
float umidade = 0.0f;
```

```
while (true) {
```

```
    if (!estado_tcp->conectado) {
```

```
        printf("Tentando conectar...\n");
```

```
        cliente_tcp_conectar(estado_tcp);
```

```
        sleep_ms(3000); // Espera 3 segundos antes de tentar de novo
```

```
    } else {
```

```
        // Se conectado, lê sensores e envia dados
```

```
        // Lê os botões (lógica invertida por causa do pull-up)
```

```
        bool botaoJoystick = !gpio_get(PINO_JOY_BOTAO);
```

```
        bool botaoA = !gpio_get(PINO_BOTAO_A);
```

```
        bool botaoB = !gpio_get(PINO_BOTAO_B);
```

```
        // Lê os eixos do Joystick
```

```

uint16_t x = ler_adc(1); // ADC 1 -> PINO_JOY_VRX
uint16_t y = ler_adc(0); // ADC 0 -> PINO_JOY_VRY

// Lê os dados do sensor DHT11
resultado_dht_t dados_dht = dht_ler_bloqueante(PINO_DHT);
if (dados_dht.valido) {
    temperatura = dados_dht.temperatura_c;
    umidade = dados_dht.umidade;
} else {
    printf("Falha na leitura do DHT11. Usando valores antigos.\n");
}

// Monta a string de dados
char mensagem[256];
snprintf(mensagem, sizeof(mensagem), "VRX=%u VRY=%u BTN=%d
A=%d B=%d TEMP=%.1f UMI=%.1f\n",
    x, y, botaoJoystick, botaoA, botaoB, temperatura, umidade);

cliente_tcp_enviar_dados(estado_tcp, mensagem);

// Espera antes da próxima leitura. O DHT11 não deve ser lido mais de
uma vez a cada 2s.
sleep_ms(2000);
}
}
}

```

## **servidor.py**

```
import asyncio
import websockets
import json
from datetime import datetime

# --- CONFIGURAÇÕES ---
PORTA_TCP = 8082
PORTA_WEBSOCKET = 8083
ARQUIVO_LOG = "log_servidor.txt"

CLIENTES_WEB_CONECTADOS = set()

def log(mensagem):
    timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_completo = f"[{timestamp}] {mensagem}"
    print(log_completo)
    with open(ARQUIVO_LOG, "a") as f:
        f.write(log_completo + "\n")

# --- FUNÇÃO CORRIGIDA ---
async def broadcast_para_web(mensagem):
    """ Envia a mensagem para todos os clientes web conectados usando
    asyncio.gather. """
    if CLIENTES_WEB_CONECTADOS:
        # Cria uma tarefa para cada corotina de envio
        tasks = [asyncio.create_task(cliente.send(mensagem)) for cliente in
CLIENTES_WEB_CONECTADOS]

        # Executa todas as tarefas de envio em paralelo
        await asyncio.gather(*tasks, return_exceptions=True)
```

```

async def manipulador_websocket(websocket, path):
    CLIENTES_WEB_CONECTADOS.add(websocket)

    log(f"Novo cliente web conectado. Total:
    {len(CLIENTES_WEB_CONECTADOS)}")

    try:
        await websocket.wait_closed()

    finally:
        CLIENTES_WEB_CONECTADOS.remove(websocket)

        log(f"Cliente web desconectou. Total:
        {len(CLIENTES_WEB_CONECTADOS)}")

async def manipulador_tcp(reader, writer):
    endereco_cliente = writer.get_extra_info('peername')
    log(f"RP2040 conectado de: {endereco_cliente}")

    try:
        while True:
            dados = await reader.readline()

            if not dados:
                log("RP2040 desconectou.")
                break

            mensagem = dados.decode('utf-8').strip()

            if not mensagem: # Ignora linhas em branco
                continue

            log(f"Recebido do RP2040: {mensagem}")

        try:
            # Verifica se a mensagem é a de boas-vindas para não tentar
            analisar

            if "Olá do RP2040!" in mensagem:

```

```
        await broadcast_para_web(json.dumps({"status": "RP2040
Conectado"})))
        continue
```

```
        dados_dict = dict(item.split('=') for item in mensagem.split(' '))
        dados_dict['VRX'] = int(dados_dict['VRX'])
        dados_dict['VRY'] = int(dados_dict['VRY'])
        await broadcast_para_web(json.dumps(dados_dict))
    except (ValueError, IndexError) as e:
        log(f"!! Erro ao analisar a mensagem: '{mensagem}', erro: {e}")
```

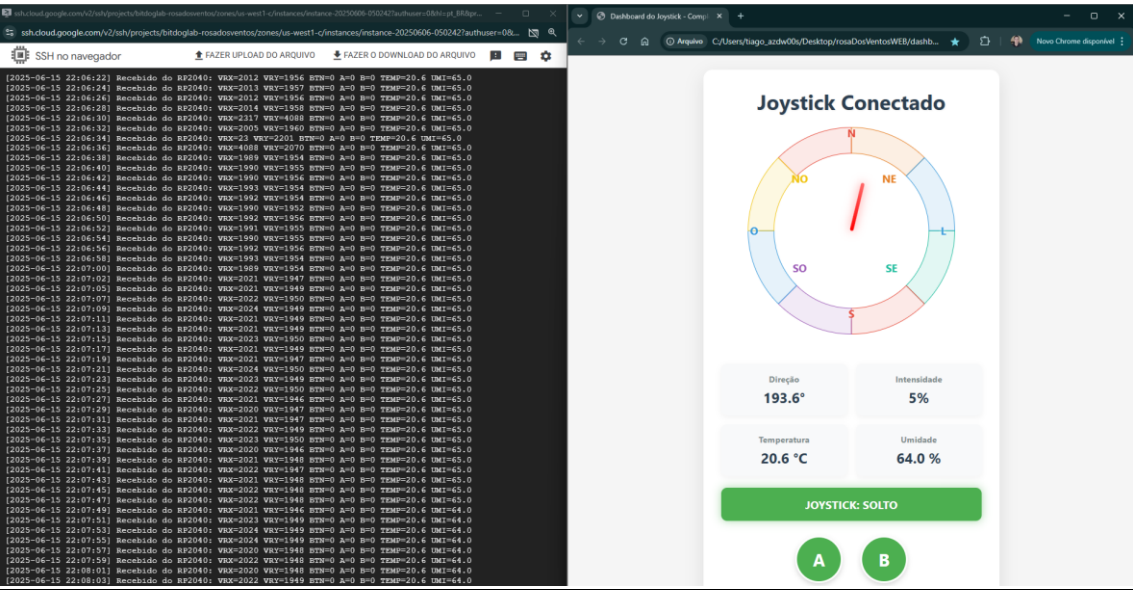
```
except Exception as e:
    log(f"!! Erro na conexão TCP: {e}")
finally:
    writer.close()
    await writer.wait_closed()
    log("Conexão com RP2040 fechada.")
```

```
async def main():
    log("Iniciando servidores...")
    servidor_tcp = await asyncio.start_server(
        manipulador_tcp, '0.0.0.0', PORTA_TCP)
    servidor_websocket = await websockets.serve(
        manipulador_websocket, "0.0.0.0", PORTA_WEBSOCKET)
    log(f"Servidor TCP rodando na porta {PORTA_TCP}")
    log(f"Servidor WebSocket rodando na porta {PORTA_WEBSOCKET}")
    await asyncio.gather(
        servidor_tcp.serve_forever(),
        servidor_websocket.serve_forever(),
    )
```

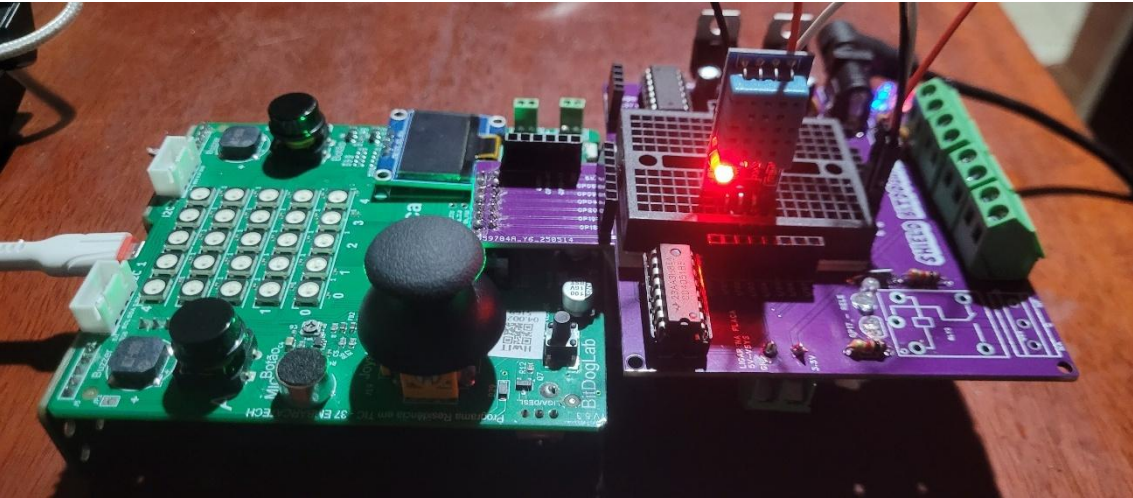
```
if __name__ == "__main__":  
    try:  
        asyncio.run(main())  
    except KeyboardInterrupt:  
        print("\nServidor encerrado manualmente.")
```



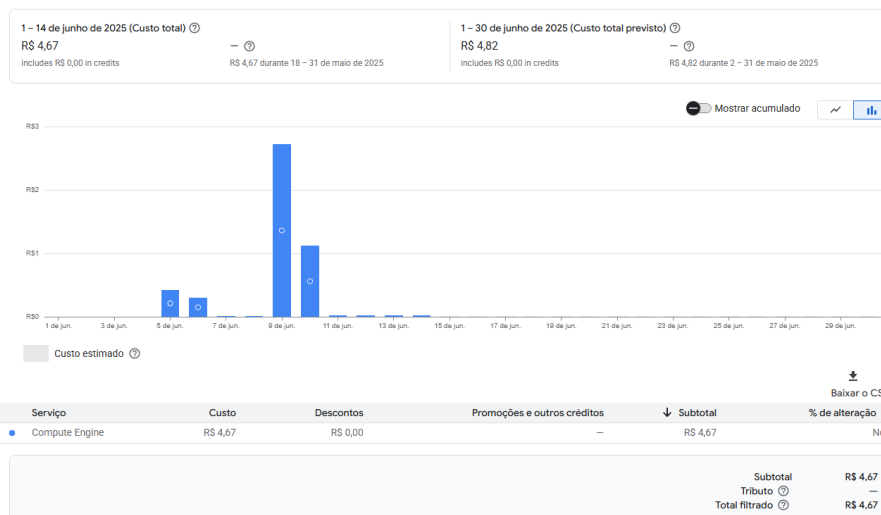
Resultados obtidos:



Valores no servidor e no navegador dashboard.html



Montagem do circuito para ligar o sensor DHT11



## Valores cobrados pelo uso do Google Cloud – Compute Engine

Durante os testes realizados com o Google Cloud Compute Engine, observei que uma máquina virtual de poucos recursos computacionais gerou um custo de quase R\$ 5,00 em apenas alguns dias, destacando que a cobrança da VM ocorre por hora, mesmo que permaneça ociosa.

Esse custo foi gerado pela arquitetura de um sistema cujo fluxo de dados é em três etapas bem definidas. Primeiramente, a placa BitDogLab realiza a coleta dos dados do sensor DHT11, formata essa informação em uma mensagem de texto padronizada e a envia para o servidor na nuvem através de uma conexão TCP. Em seguida, o servidor na nuvem, atuando como uma ponte, recebe a mensagem enviada pela placa e a retransmite imediatamente em tempo real para a interface do dashboard, utilizando o protocolo WebSocket. Finalmente, a interface no navegador recebe a mensagem de texto, analisa seu conteúdo para extrair os valores numéricos e atualiza os gráficos e mostradores com as novas informações do sensor.

Link do GitHub:

[https://github.com/tiagocopelli/EmbarcaTech\\_Aplicacoes\\_com\\_comunicacao\\_s  
em\\_fio\\_para\\_IoT](https://github.com/tiagocopelli/EmbarcaTech_Aplicacoes_com_comunicacao_s_em_fio_para_IoT)