CAPÍTULO 5

FIRMWARE C++ PARA ESP32: ARQUITETURA DE CLIENTE MAGRO, SINCRONIZAÇÃO HTTP E ACIONAMENTO DE ATUADORES NA IRRIGAÇÃO IoT

1. INTRODUÇÃO E INSERÇÃO DO SISTEMA EMBARCADO NO AMBIENTE AGRÍCOLA

No paradigma da agricultura digital e da Internet das Coisas (Internet of Things – IoT), os microcontroladores de borda atuam como a ponte física indispensável entre o mundo digital (onde residem os algoritmos e servidores) e a lavoura (onde a água deve ser fisicamente aplicada ao solo). Enquanto os capítulos anteriores exploraram a construção das interfaces gráficas web (Capítulo 1), as especificações hidráulicas e eletromecânicas dos atuadores (Capítulo 2), a instrumentação eletrônica de controle e leitura (Capítulo 3) e o protocolo de comunicação HTTP REST (Capítulo 4), este capítulo dedica-se integralmente à inteligência embarcada que executa diretamente no microcontrolador ESP32.

O ESP32 é um sistema em chip (*System on a Chip* – SoC) de baixo custo, elevado desempenho e baixo consumo energético, dotado de conectividade Wi-Fi e Bluetooth nativas. No projeto hipotético de irrigação automatizada analisado nesta obra, o ESP32 fica instalado no painel elétrico de campo, posicionado estrategicamente próximo ao conjunto motobomba e ao coletor hidráulico de distribuição. Sua função principal é executar um firmware escrito na linguagem C++ que se conecta à rede sem fio da propriedade rural, consulta periodicamente uma API web remota e aciona os relés acoplados às válvulas solenoides e à motobomba principal.

Para que agrônomos, técnicos agrícolas e engenheiros de software trabalhem em harmonia, é preciso compreender que o código-fonte C++ que roda no microcontrolador não é apenas um script de controle: ele é o agente executor das decisões agronômicas. Cada linha de código relacionada ao consumo de dados via HTTP traduz a intenção de irrigar ou suspender a rega em respostas elétricas de 3,3 V nos pinos de saída (*General Purpose Input/Output* – GPIO).

2. A CONCEITUAÇÃO DE CLIENTE MAGRO (*THIN CLIENT*) SEGUNDO TANENBAUM

A estruturação do software embarcado no ESP32 segue os preceitos clássicos das arquiteturas distribuídas descritas pelo renomado cientista da computação Andrew S. Tanenbaum. Em sua classificação de modelos cliente-servidor, Tanenbaum estabelece a distinção entre clientes pesados (*Thick/Fat Clients*) e clientes magros (*Thin Clients*) com base na distribuição das camadas de apresentação, lógica de negócio e armazenamento de dados.

Um cliente pesado concentra localmente a inteligência de processamento, as regras de negócio e, frequentemente, a tomada de decisão. Caso o ESP32 fosse projetado como um cliente pesado, ele precisaria armazenar internamente as tabelas de coeficientes de

cultura (Kc), as equações de evapotranspiração de Penman-Monteith, os relógios de tempo real (RTC) com os horários de início e fim de irrigação para cada talhão, e os históricos de amostragem. No entanto, essa abordagem traria sérias limitações para cenários agrícolas rurais:

- **Complexidade de Manutenção e Atualização:** Qualquer alteração no manejo agronômico (como mudar o tempo de rega do setor do melão de 15 para 20 minutos) exigiria que um técnico fosse até o campo e reprogramasse fisicamente o microcontrolador via cabo USB ou atualizações de firmware over-theair (OTA).
- **Restrição de Recursos de Hardware:** Microcontroladores possuem memória RAM limitada (o ESP32 dispõe de 520 KB de SRAM interna). Executar cálculos complexos e armazenar grandes volumes de dados comprometeria a estabilidade do dispositivo.
- **Risco de Dessincronização:** Múltiplos microcontroladores em setores distintos da fazenda poderiam operar com relógios descalibrados ou regras desatualizadas, gerando conflitos no uso da bomba hidráulica principal.

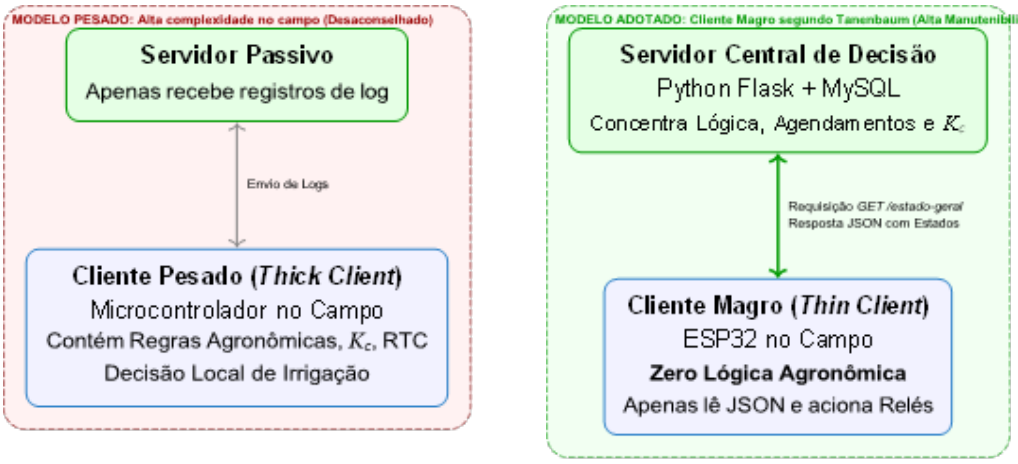
Para contornar as limitações de hardware, o firmware desenvolvido espelha o modelo de **Cliente Magro** (*Thin Client*) (TANENBAUM; STEEN, 2017) adaptado à arquitetura de borda e nuvem (SHI; CAO et al., 2016; SATYANARAYANAN, 2017). Em uma versão simplificada, o dispositivo de borda realiza apenas a interface de entrada/-saída, enquanto toda a tomada de decisão agronômica reside no servidor central. No entanto, é fundamental destacar que sistemas IoT profissionais raramente utilizam dispositivos de borda desprovidos de lógica autônoma. Visando resiliência, a computação de borda moderna orienta a manutenção de regras mínimas de segurança local, cronômetros de vigilância (*watchdogs*), limites operacionais e lógicas elementares no microcontrolador, garantindo que o sistema atue proativamente mesmo diante de prolongadas falhas na rede de acesso global (SHI; DUSTDAR, 2016; BAZZI et al., 2023).

Na prática da irrigação IoT, o ESP32 operando como cliente magro não sabe se o solo está seco ou úmido, não calcula a lâmina de água e não possui conhecimento sobre agendamentos futuros. O ESP32 simplesmente limita-se a:

1. Conectar-se à rede Wi-Fi local;
2. Disparar uma requisição HTTP do tipo GET para o servidor central Flask a cada 5 segundos;
3. Receber a resposta contendo a decisão já tomada pelo servidor em formato JSON;
4. Alterar o estado dos pinos GPIO (LIGADO/DESLIGADO) de acordo com os valores booleanos recebidos.

A Figura 1 ilustra a comparação conceitual entre as abordagens de cliente pesado e cliente magro de Tanenbaum aplicadas à automação agrícola.

Figura 15: Comparação arquitetural segundo a taxonomia de Tanenbaum: no modelo de Cliente Magro adotado, o ESP32 limita-se a executar acionamentos elétricos instruídos pelo servidor central.



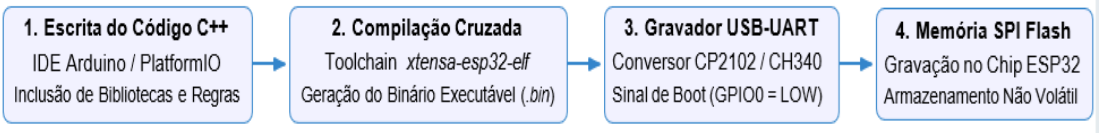
Essa descentralização traz vantagens expressivas para o produtor rural: se o agrônomo Thiago decidir reconfigurar todos os parâmetros de irrigação do cultivo de tomate, a alteração é efetuada instantaneamente na interface web. No próximo ciclo de consulta (5 segundos depois), o ESP32 receberá as novas ordens de acionamento sem necessidade de qualquer intervenção técnica no hardware de campo.

3. FLUXO DE DESENVOLVIMENTO, COMPILAÇÃO E GRAVAÇÃO (FLASHING) NO ESP32

Para que o código C++ escrito no computador passe a executar de maneira autônoma no chip ESP32 instalado na fazenda, ele deve submeter-se a um fluxo estruturado de engenharia de software embarcado. Esse ciclo compreende as etapas de escrita, compilação cruzada, transferência serial para a memória Flash e inicialização do microcontrolador.

A Figura 2 apresenta a esteira completa do fluxo de subida (flashing) do programa no ESP32.

Figura 16: Fluxo de compilação cruzada e gravação do firmware C++ no microcontrolador ESP32.



As etapas do processo de gravação desdobram-se conforme detalhado a seguir:

1. **Ambiente de Desenvolvimento (IDE):** O programador de TI escreve o código C++ utilizando um ambiente como a Arduino IDE ou o VS Code com a extensão PlatformIO. O código faz referência às bibliotecas nativas de comunicação sem fio e serialização de dados.
2. **Compilação Cruzada (*Cross-Compilation*):** Como o computador de desenvolvimento

possui arquitetura x86/x64 e o ESP32 utiliza microarquitetura Xtensa LX6 de 32 bits, a IDE utiliza um compilador cruzado (*xtensa-esp32-elf-g++*). O compilador traduz as instruções em C++ para linguagem de máquina executável e gera um arquivo binário comprimido com extensão .bin.

3. **Comunicação Serial e Protocolo de Bootloader:** O computador é conectado à placa ESP32 por meio de um cabo USB. Um chip conversor USB-para-UART (como o CP2102 ou CH340 instalado na placa) converte os pacotes USB em sinais seriais TTL (transmissão TX e recepção RX). Para iniciar a gravação, o circuito de autorreset coloca o pino GPIO0 em nível lógico baixo (0 V) enquanto aplica um pulso no pino EN (Reset), forçando a ROM do ESP32 a entrar no modo de gravação (Download Bootloader).
4. **Gravação na Memória SPI Flash:** O programa utilitário (esptool.py) envia os blocos do arquivo binário através da interface serial a uma taxa de transmissão típica de 921.600 baud. Os dados são gravados permanentemente na memória Flash externa do ESP32. Por ser uma memória não volátil, o código-fonte permanece salvo mesmo quando a alimentação elétrica da fazenda é desligada.
5. **Inicialização Autônoma:** Concluída a gravação, o pino EN é liberado, o ESP32 reinicia em modo de execução normal, carrega o código da Flash para a memória SRAM e executa automaticamente a função setup(), seguida do loop infinito loop().

4. DEPENDÊNCIAS DE SOFTWARE E BIBLIOTECAS DA PLATAFORMA EMBARCADA

O funcionamento do cliente magro ESP32 fundamenta-se em três bibliotecas principais do ecossistema Arduino/ESP32. O entendimento do papel de cada uma dessas dependências é essencial para a correta compilação e operação do sistema:

4.1. Biblioteca *WiFi.h*

A biblioteca *WiFi.h* é a pilha de protocolos de comunicação sem fio nativa do núcleo ESP32 para Arduino. Ela abstrai a complexidade do hardware de radiofrequência de 2,4 GHz e fornece métodos em C++ para gerenciar o estado da conexão de rede. Suas principais responsabilidades no sistema de irrigação incluem:

- Configurar o modo de operação do chip como Estação Wi-Fi (*WIFI_STA*), permitindo que o ESP32 se conecte ao roteador da fazenda como um dispositivo cliente.
- Negociar o aperto de mão (*handshake*) de segurança utilizando criptografia WPA2-PSK.
- Gerenciar a atribuição dinâmica de endereço IP via DHCP (*Dynamic Host Configuration Protocol*).
- Fornecer a função de sondagem de estado *WiFi.status()*, que retorna a constante

WL_CONNECTED quando a conexão com o ponto de acesso está ativa e funcional.

4.2. Biblioteca *HTTPClient.h*

A biblioteca *HTTPClient.h* implementa a camada de aplicação HTTP/HTTPS sobre a pilha TCP/IP do ESP32. Ela permite que o microcontrolador funcione como um cliente web capaz de abrir soquetes de rede, formatar requisições HTTP REST e ler respostas transmitidas por servidores remotos. Dentre suas funções fundamentais, destacam-se:

- Estabelecer a conexão com a URL de destino através do método *begin(serverName)*.
- Executar transações HTTP como requisições GET (*http.GET()*) ou POST (*http.POST()*).
- Negociar a camada de segurança TLS/SSL para conexões criptografadas HTTPS (porta 443).
- Capturar os códigos de status de retorno HTTP (como 200 OK ou 404 Not Found) e extrair o corpo da resposta em texto simples através do método *getString()*.

4.3. Biblioteca *ArduinoJson.h*

A biblioteca *ArduinoJson* (desenvolvida por Benoît Blanchon) é a biblioteca padrão para serialização e desserialização de documentos JSON em sistemas embarcados. Devido à rigidez de memória dos microcontroladores, a *ArduinoJson* se destaca por gerenciar a memória de forma extremamente eficiente, evitando fragmentação na SRAM. Suas atribuições no firmware incluem:

- Alocar um documento de memória otimizado através da classe *JsonDocument*.
- Converter a string texto obtida do servidor em uma estrutura de dados de árvore pesquisável em C++ por meio do método *deserializeJson(doc, payload*).
- Validar a integridade estrutural da mensagem recebida através do objeto de verificação de erros *DeserializationError.*
- Permitir o acesso a arrays (*JsonArray*) e objetos (*JsonObject*) com rápida conversão de tipos de dados (booleanos, inteiros e strings).

5. APRESENTAÇÃO E ANÁLISE DO CÓDIGO-FONTE C++

O código C++ completo responsável por transformar o ESP32 no cliente magro de controle de irrigação é apresentado no arquivo de listagem a seguir.

Listing 1: Código C++ completo para o microcontrolador ESP32: sincronização via HTTP e acionamento de relés.

```
#include <WiFi.h> #include
<HTTPClient.h> #include
<ArduinoJson.h>

#define LIGADA HIGH #define
DESLIGADA LOW #define
BOMBA 22

#define SOLENOIDE_1 23
#define SOLENOIDE_2 21
#define SOLENOIDE_3 32
#define SOLENOIDE_4 18


const char *ssid = "Tenda_9519E8";
const char *password = "able7436";
const char *serverName = "https://sergiosouzanovak.com.br/backend-irrigacao/estado-geral";


const int reles[5] = {SOLENOIDE_1, SOLENOIDE_2, SOLENOIDE_3, SOLENOIDE_4, BOMBA};


void desligarTudo() {
  for (int i = 0; i < 5; i++) {
    digitalWrite(reles[i], DESLIGADA);
  }
}


void setup() {
  Serial.begin(115200);


  for (int i = 0; i < 5; i++) { pinMode(reles[i],
    OUTPUT); digitalWrite(reles[i],
```

```
36    while (WiFi.status() != WL_CONNECTED) {
37      delay(500);
38      Serial.print(".");
39    }
40
41    Serial.println("\nConectado!");
42  }
43
44  void loop() {
45
46    if (WiFi.status() == WL_CONNECTED) {
47
48      HTTPClient http;
49      http.begin(serverName);
50
51      int httpResponseCode = http.GET();
52
53      if (httpResponseCode > 0) {
54
55        String payload = http.getString();
56        Serial.println("JSON: " + payload);
57
58        JsonDocument doc;
59        DeserializationError error = deserializeJson(doc, payload);
60
61        if (error) {
62
63          Serial.println("erro requisicao");
64          desligarTudo();
65
66        } else {
67
68          JsonArray solenoides = doc["solenoides"].as<JsonArray>();
69
70          for (int i = 0; i < 4; i++) {
71
72            int idSolenoid = i + 1;
73            bool ativo = false;
74
75            for (JsonObject sol : solenoides) {
76              if (sol["id"].as<int>() == idSolenoid) {
77                ativo = sol["ativo"].as<bool>();
```

```
78                    break;
79                  }
80                }
81
82                digitalWrite(reles[i], ativo ? LIGADA : DESLIGADA);
83
84                Serial.printf(
85                  "Solenoide %d (GPIO %d): STATUS %s\n",
86                  idSolenoid,
87                  reles[i],
88                  ativo ? "LIGADA" : "DESLIGADA"
89                );
90              }
91
92              bool bombaAtiva = doc["bomba"].as<bool>();
93              digitalWrite(reles[4], bombaAtiva ? LIGADA : DESLIGADA);
94
95              Serial.printf("Bomba (GPIO %d): STATUS %s\n", BOMBA, bombaAtiva ? "LIGADA" : "DESLIGADA");
96            }
97
98          } else {
99
100           Serial.println("erro requisicao");
101           desligarTudo();
102         }
103
104         http.end();
105
106       } else {
107
108         Serial.println("erro requisicao");
109         desligarTudo();
110       }
111
112     delay(5000);
113   }
```

6. ESTRUTURA DE CONSTANTES, MAPEAMENTO DE GPIOS E PARAMETRIZAÇÃO DE REDE

A primeira seção do código (linhas 1 a 16 da Listagem 1) estabelece as fundações operacionais do firmware, mapeando os pinos elétricos do microcontrolador e registrando os parâmetros de conectividade.

6.1. Diretivas de Mapeamento de Pinos *(#define)*

O código utiliza diretivas do pré-processador C++ para associar nomes amigáveis às constantes de estado e aos pinos físicos de entrada e saída (GPIOs):

- *#define LIGADA HIGH* e *#define DESLIGADA LOW:* Definem que o estado LIGADA corresponde ao nível lógico alto (3,3 V na saída do ESP32) e o estado DES-LIGADA ao nível lógico baixo (0 V). Essa abstração evita erros de lógica e facilita a leitura do código por profissionais de agronomia.
- *#define BOMBA 22*: Associa o controle da motobomba principal ao pino digital GPIO 22.
- *#define SOLENOIDE_1 23, SOLENOIDE_2 21, SOLENOIDE_3 32 e SOLENOIDE_4 18:* Mapeiam os quatro setores hidráulicos de irrigação da propriedade às respetivas portas GPIO do ESP32.

6.2. Parâmetros da Rede Wi-Fi e Endpoint HTTPS

As constantes de ponteiro de caractere armazenam as credenciais de autenticação e a URL do servidor central:

- *const char *ssid* = "*Tenda_9519E8*": O identificador do ponto de acesso Wi-Fi instalado no galpão agrícola da fazenda.
- *const char *password* = "*able7436*": A chave de segurança para associação à rede sem fio.
- *const char *serverName* = https://sergiosouzanovak.com.br/backend-irrigacao/estado-geral: O endereço HTTPS completo da API REST que o ESP32 consultará a cada ciclo para obter o estado dos atuadores.

6.3. Vetor de Reorganização de Atuadores

A instrução *const int reles[5] = {SOLENOIDE_1, SOLENOIDE_2, SOLENOIDE_3, SO-LENOIDE_4, BOMBA}* agrupa as constantes de pinos em uma estrutura indexada de vetor (array). Essa abordagem é uma excelente prática de programação embarcada, pois permite que o firmware percorra todos os atuadores da lavoura utilizando laços de repetição (for), reduzindo drasticamente o tamanho do código compilado e simplificando a manutenção.

7. MECANISMO DE SEGURANÇA E SALVAGUARDA HIDRÁULICA: A FUNÇÃO *desligarTudo()*

Dentre todos os blocos do firmware, a função *desligarTudo()* (linhas 18 a 22 da Listagem 1) reveste-se de importância crítica para a segurança agronômica e patrimonial da propriedade rural.

```
void desligarTudo() {
  for (int i = 0; i < 5; i++) {
    digitalWrite(reles[i], DESLIGADA);
  }
}
```

Seu funcionamento consiste em iterar pelos cinco elementos do vetor de relés e aplicar o nível lógico *DESLIGADA* (0 V) a cada uma das portas GPIO, forçando a abertura imediata dos contatos dos relés e desenergizando os solenoides e a motobomba.

7.1. O Conceito de Falha Segura (*Fail-Safe*) na Agricultura

Na engenharia de controle e automação, um sistema é classificado como Falha Segura (*Fail-Safe*) quando, diante de qualquer anomalia ou colapso na infraestrutura, ele transiciona automaticamente para um estado passivo inofensivo, impedindo catástrofes físicas ou econômicas.

Na irrigação agrícola automatizada, a ocorrência de falhas é um evento inevitável devido às intempéries do ambiente rural:

- O sinal da rede Wi-Fi pode sofrer degradação temporária por interferência atmosférica ou obstrução por máquinas agrícolas;
- O servidor backend Flask pode ficar inacessível por queda no fornecimento de energia da cidade;
- O roteador da fazenda pode reiniciar por sobreaquecimento;
- O pacote de dados JSON retornado pela API pode ser corrompido durante a transmissão.

Se o microcontrolador ESP32 não possuísse o mecanismo de falha segura da função *desligarTudo()*, qualquer queda de conexão corrida no momento em que a bomba estivesse ligada faria com que o sistema mantivesse os relés energizados indefinidamente. As consequências agronômicas e financeiras dessa omissão seriam severas:

1. **Encharcamento e Anoxia Radicular:** A aplicação ininterrupta de água saturaria o perfil do solo, expulsando o oxigênio dos macroporos, provocando a asfixia

do sistema radicular das plantas e o surgimento de doenças fúngicas (como *Phytophthora* spp.).

2. **Rompimento de Tubulações Hidráulicas:** Se os solenoides fossem fechados mas a motobomba permanecesse ligada sem vazão, a pressão manométrica da rede ultrapassaria os limites dos tubos de PVC, provocando o estouro de adutoras.
3. **Cavitação e Queima da Motobomba:** O funcionamento continuado da motobomba sem circulação renovada de água causaria o superaquecimento do fluido interno, destruição dos selos mecânicos e queima do enrolamento elétrico do motor de 5,5 CV.
4. **Desperdício Hídrico e Energético:** Dezenas de milhares de litros de água e quilowatts-hora de energia elétrica seriam consumidos inultilmente, elevando os custos operacionais do produtor.

Na lógica implementada para fins didáticos, a função *desligarTudo()* é acionada como uma aproximação de falha segura em todos os caminhos de exceção do firmware: perda de Wi-Fi, falha HTTP ou JSON corrompido (INTERNATIONAL ELECTROTE-CHNICAL COMMISSION, 2016). Contudo, em cenários industriais e agrícolas de grande porte, o conceito de falha segura deve obrigatoriamente derivar de uma análise formal de perigos e matriz de riscos (LEVESON, 2011). Desligar todos os atuadores subitamente nem sempre representa o estado mais seguro; certos processos exigem válvulas normalmente abertas para alívio de pressão hidráulica ou intertravamentos específicos previstos em normas de segurança funcional (INTERNATIONAL ELECTROTECHNI-CAL COMMISSION, 2010; INTERNATIONAL ORGANIZATION FOR STANDAR-DIZATION, 2010). Assim, embora a abordagem de contingência total proteja o sistema simplificado aqui concebido, sua escalabilidade exige a avaliação criteriosa de modos degradados e tolerância a falhas (BENNETT, 1994).

8. PROCEDIMENTO DE INICIALIZAÇÃO: A FUNÇÃO *setup()*

A função *setup()* (linhas 24 a 40 da Listagem 1) é executada exatamente uma única vez, logo que o ESP32 é energizado ou após a ocorrência de um sinal de reset. Ela é responsável por preparar a infraestrutura interna do microcontrolador antes da entrada no ciclo de operação continuada.

1. **Inicialização do Monitor Serial:** A instrução *Serial.begin(115200*) configura a porta de comunicação serial assíncrona para operar a uma taxa de transmissão de 115.200 bits por segundo (baud rate). Isso possibilita que o técnico de TI conecte um computador ao ESP32 e monitore em tempo real os relatórios de diagnóstico emitidos pelo firmware.
2. **Configuração de Direção dos Pinos (*pinMode*):** O laço for percorre as portas digi-

tais associadas aos relés e executa a função *pinMode(reles[i], OUTPUT*), instruindo o hardware interno do ESP32 a configurar esses pinos como saídas digitais capazes de fornecer corrente para os optoacopladores das placas de relé.

3. **Garantia de Estado Inicial Desenergizado:** Imediatamente após definir os pinos como saída, a instrução *digitalWrite(reles[i], DESLIGADA)* força todos os relés ao nível 0 V. Isso evita o acionamento indesejado dos motores durante a etapa de energização do painel elétrico.
4. **Associação Wi-Fi e Espera Bloqueante:** A chamada *WiFi.begin(ssid, password)* dispara a solicitação de conexão à rede sem fio da propriedade. Em seguida, o programa entra em um laço de repetição bloqueante while(WiFi.status()!=WL_ CONNECTED), no qual o ESP32 aguarda a confirmação da conexão, emitindo um ponto na interface serial a cada 500 milissegundos (*delay(500)*). Assim que a associação é concluída, a mensagem "*Conectado*!" é exibida na tela.

9. O CICLO INFINITO DE POLLING E CONSUMO DA API: A FUNÇÃO *loop()*

Após o término da função *setup()*, o controle de execução transita para a função *loop()* (linhas 42 a 110 da Listagem 1), que é executada continuamente de forma repetitiva enquanto o microcontrolador permanecer energizado.

A Figura 3 ilustra a máquina de estados completa e o fluxograma de decisões executado dentro da função *loop()*.

9.1. Verificação de Conectividade Wi-Fi

No início de cada iteração, a *instrução if (WiFi.status() == WL_CONNECTED)* testa se a conexão sem fio permanece ativa. Se o roteador tiver caído ou a placa tiver perdido o sinal, o programa desvia diretamente para o bloco else, emitindo a mensagem de erro no monitor serial e executando a função de segurança *desligarTudo().*

9.2. Disparo da Requisição HTTP GET

Estando a rede Wi-Fi operacional, o firmware instancia um objeto da classe *HTTPClient* e associa o endereço HTTPS de destino através de *http.begin(serverName).* A chamada bloqueante *int httpResponseCode = http.GET()* faz com que o ESP32 monte o pacote HTTP, abra a conexão TCP na porta 443, negocie a criptografia TLS e aguarde o retorno do servidor central Flask.

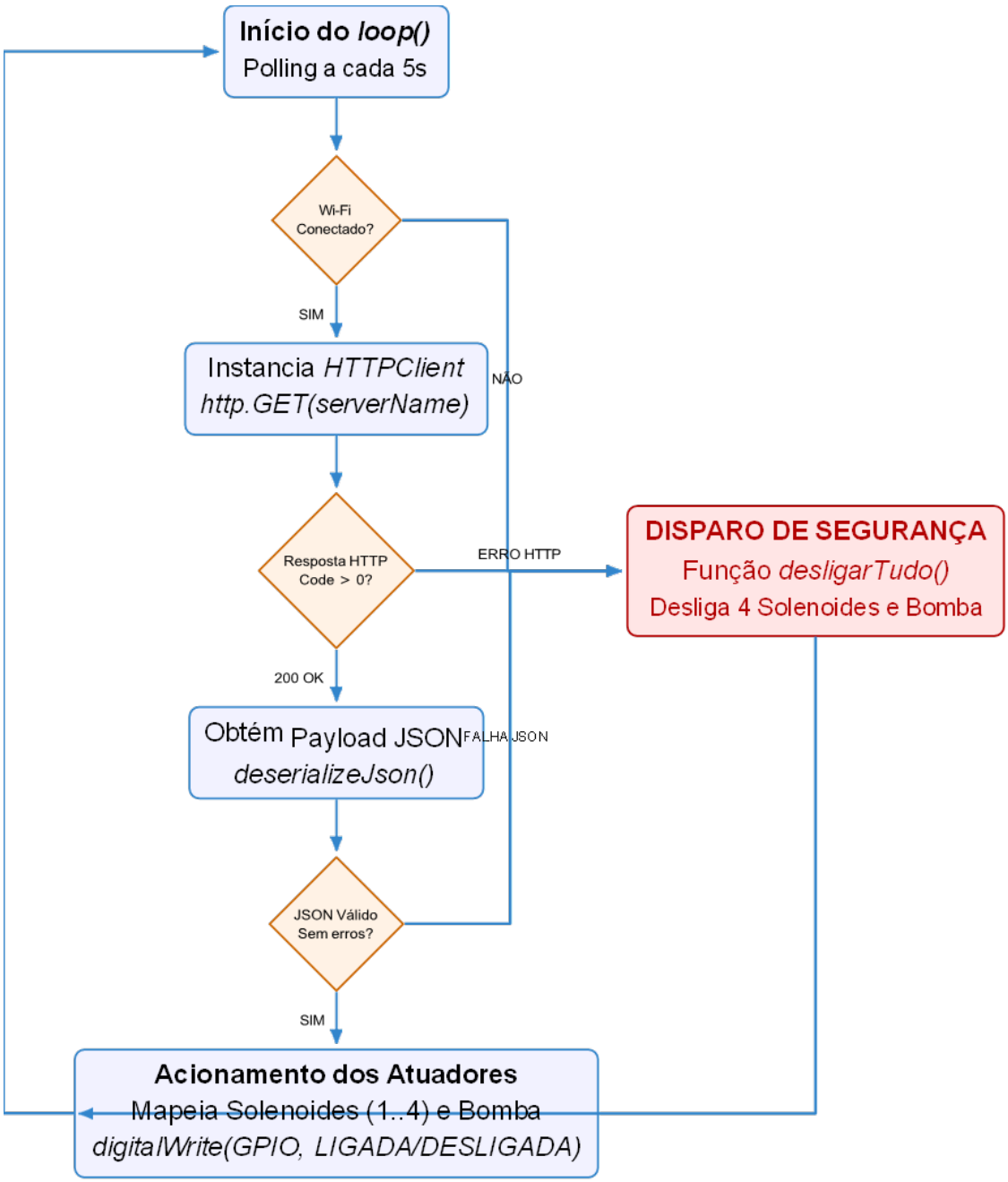
Se a comunicação for bem-sucedida, o código de status retornado é maior que zero (tipicamente 200 OK). Caso contrário (se o servidor estiver fora do ar ou o nome de domínio não puder ser resolvido pelo DNS), o código retornado é um valor negativo e o firmware aciona imediatamente o bloco de segurança *desligarTudo().*

9.3. Desserialização e Parsing do Documento JSON

Ao confirmar o recebimento de uma resposta HTTP válida, o ESP32 extrai o texto do

corpo da mensagem por meio de String payload = http.getString(). Em seguida, o firmware aloca uma instância de JsonDocument e executa o método deserializeJson(doc,

Figura 17: Fluxograma de decisões e salvaguardas da função *loop()* do ESP32.



O parâmetro de verificação de erro *DeserializationError* error analisa se a string recebida obedece rigorosamente à sintaxe JSON. Se o pacote tiver chegado incompleto ou corrompido, a condição if (error) é satisfeita, acionando a função *desligarTudo()* para evitar que dados truncados levem a acionamentos erráticos de solenoides no campo.

9.4. Mapeamento Dinâmico dos Setores e Acionamento Elétrico

Se a desserialização JSON for concluída com sucesso, o firmware

inicia o processo de interpretação dos dados recebidos. A resposta típica emitida pela API do sistema de irrigação apresenta o seguinte formato:

```
{
  "bomba": true,
  "solenoides": [
    { "id": 1, "ativo": true },
    { "id": 2, "ativo": false },
    { "id": 3, "ativo": false },
    { "id": 4, "ativo": false }
  ]
}
```

O código acessa o vetor de solenoides através da instrução *JsonArray solenoides = doc[“solenoides”*].as<JsonArray>(). Em seguida, um laço de repetição for (int i = 0; i < 4; i++) percorre as quatro válvulas cadastradas:

```
for (int i = 0; i < 4; i++) {
  int idSolenoid = i + 1;
  bool ativo = false;

  for (JsonObject sol : solenoides) {
    if (sol["id"].as<int>() == idSolenoid) {
      ativo = sol["ativo"].as<bool>();
      break;
    }
  }

  digitalWrite(reles[i], ativo ? LIGADA : DESLIGADA);
}
```

Para cada setor da lavoura (1 a 4), o algoritmo busca a correspondência exata do atributo “*id*” dentro do array JSON. Ao localizar o objeto correspondente, a variável booleana *ativo* recebe o valor presente na chave “*ativo*”.

A instrução ternária *digitalWrite(reles[i], ativo ? LIGADA : DESLIGADA*) aplica diretamente a decisão: se ativo for verdadeiro (*true*), a saída GPIO física do ESP32 é colocada em 3,3 V (*HIGH*), acionando a bobina do relé e abrindo a válvula solenoide correspondente na tubulação de irrigação. Caso contrário, a saída é mantida em 0 V (*LOW*), fechando a válvula.

9.5. Acionamento da Motobomba Principal e Intervalo de Polling

Após atualizar o estado das quatro solenoides, o firmware lê o atributo booleano correspondente à bomba principal:

```
1 bool bombaAtiva = doc["bomba"].as<bool>();
2 digitalWrite(reles[4], bombaAtiva ? LIGADA : DESLIGADA);
```

A instrução aplica a leitura diretamente na porta GPIO 22 *(reles[4]),* energizando ou desenergizando a bobina da contatora elétrica de potência que alimenta a motobomba de 5,5 CV.

Por fim, a função *http.end()* encerra formalmente a conexão do soquete HTTP, liberando os recursos de memória RAM do ESP32, e a instrução *delay(5000)* suspende a execução por 5.000 milissegundos (5 segundos) antes que o próximo ciclo da função *loop()* seja iniciado.

10. CONSIDERAÇÕES PRÁTICAS E ENGENHARIA DE IMPLANTAÇÃO NO CAMPO

A implementação com sucesso de um cliente magro ESP32 na agricultura de precisão exige a observância de boas práticas de engenharia eletrônica e de redes rurais:

- **Estabilidade de Alimentação Elétrica:** Motores e solenoides geram ruídos eletromagnéticos e surtos de tensão ao serem acionados. O ESP32 deve ser alimentado por uma fonte chaveada regulada independente de 5 V / 2 A, acompanhada de capacitores de desacoplamento e diodos de roda livre (*flyback*) instalados nas bobinas dos relés para evitar autorresets provocados por transientes elétricos.
- **Qualidade de Sinal Sem Fio (RSSI):** Como o cliente magro depende totalmente da comunicação com o servidor para irrigar, o painel elétrico de campo deve possuir antena Wi-Fi externa com ganho adequado (ex: antena omnidirecional de 5 dBi) para garantir uma relação sinal-ruído robusta em relação ao roteador do galpão.
- **Certificados de Segurança em Conexões HTTPS:** A utilização da URL com pro-tocolo criptografado *https://* exige atenção ao gerenciamento de certificados SSL/-TLS no ESP32. Em sistemas de produção, deve-se carregar a chave do certificado de autoridade certificadora (CA) ou utilizar validação de impressão digital (*finger-print*) para impedir ataques de interceptação na rede sem fio da fazenda.

11. SÍNTESE DO CAPÍTULO E INTEGRAÇÃO DO SISTEMA DE IRRIGAÇÃO

O desenvolvimento do firmware C++ para o ESP32 consolida o modelo de arquitetura distribuída de cliente magro proposto por Tanenbaum. Ao transferir a inteligência agronômica, o cálculo das lâminas de água e as tabelas de agendamento para o servidor central Flask/MySQL e manter o microcontrolador dedicado exclusivamente à sincronização HTTP e acionamento elétrico dos relés, o sistema atinge os mais elevados padrões de manutenibilidade, segurança e flexibilidade operacional.

Com este capítulo, encerra-se a fundamentação completa da obra: a integração harmônica entre as interfaces gráficas web (Capítulo 1), a infraestrutura de sensores e atuadores (Capítulo 2), os circuitos eletrônicos de instrumentação e acionamento (Capítulo 3), o protocolo de comunicação HTTP REST (Capítulo 4) e o código embarcado em C++ no microcontrolador ESP32 (Capítulo 5). Essa união entre a Ciência da Computação e a Engenharia Agronômica estabelece as bases definitivas para a construção de sistemas de irrigação automatizados de alta performance na lavoura digital.

REFERÊNCIAS

BAZZI, Alessandro et al. (Ed.). **Device-Edge-Cloud Continuum: Paradigms, Architectures and Applications**. [S.l.]: Springer, 2023. DOI: 10.1007/978-3-031-42194-5.

BENNETT, Stuart. **Real-Time Computer Control: An Introduction**. New York: Prentice Hall, 1994.

INTERNATIONAL ELECTROTECHNICAL COMMISSION. **Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems**. [S.l.: s.n.], 2010.

INTERNATIONAL ELECTROTECHNICAL COMMISSION. **Safety of Machinery: Electrical Equipment of Machines – Part 1: General Requirements**. [S.l.: s.n.], 2016.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **Safety of Machinery: General Principles for Design – Risk Assessment and Risk Reduction**. [S.l.: s.n.], 2010.

LEVESON, Nancy G. **Engineering a Safer World: Systems Thinking Applied to Safety**. Cambridge: MIT Press, 2011.

SATYANARAYANAN, Mahadev. The emergence of edge computing. **Computer**, v. 50, n. 1, p. 30–39, 2017. DOI: 10.1109/MC.2017.9.

SHI, Weisong; CAO, Jie et al. Edge computing: Vision and challenges. **IEEE Internet of Things Journal**, v. 3, n. 5, p. 637–646, 2016. DOI: 10.1109/JIOT.2016.2579198.

SHI, Weisong; DUSTDAR, Schahram. The promise of edge computing. **Computer**, v. 49, n. 5, p. 78–81, 2016. DOI: 10.1109/MC.2016.145.

TANENBAUM, Andrew S.; STEEN, Maarten van. **Distributed Systems: Principles and Paradigms**. 3. ed. Upper Saddle River: Pearson, 2017.