UNIVERSIDADE ESTADUAL PAULISTA

“JÚLIO DE MESQUITA FILHO”

Faculdade de Engenharia e Ciências de Guaratinguetá

PEDRO THOMAS HOMEM DE MELO MENDES

Desenvolvimento de uma API RESTful para sistemas de automação residencial

Guaratinguetá
2024

Pedro Thomas Homem de Melo Mendes

Desenvolvimento de uma API RESTful para sistemas de automação residencial

Trabalho de Graduação apresentado ao Conselho de Curso de Graduação em Engenharia Elétrica da Faculdade de Engenharia e Ciências do Campus de Guaratinguetá, Universidade Estadual Paulista, como parte dos requisitos para obtenção do diploma de Graduação em Engenharia Elétrica.

Orientador (a): Prof. M.Sc. Thiago José Michelin
Coorientador (a): Prof. Dr. Daniel Julien Barros da Silva Sampaio

Guaratinguetá
2024

M538d

Mendes, Pedro Thomas Homem de Melo
Desenvolvimento de uma API RESTful para sistemas de automação residencial / Pedro Thomas Homem de Melo mendes - Guaratinguetá, 2024.
120 f : il.
Bibliografia: f. 87-89

Trabalho de Graduação em Engenharia Elétrica – Universidade Estadual Paulista, Faculdade de Engenharia e Ciências de Guaratinguetá, 2024.
Orientador: Prof. M.Sc. Thiago José Michelin
Coorientador: Prof. Dr. Daniel Julien B. S. Sampaio

1. Automação residencial. 2. Controle eletrônico.
3. Microcontroladores. I. Título.

CDU 681.527.7

Luciana Máximo
Bibliotecária/CRB-8 3595

UNIVERSIDADE ESTADUAL PAULISTA

"JÚLIO DE MESQUITA FILHO"

Faculdade de Engenharia e Ciências de Guaratinguetá

PEDRO THOMAS HOMEM DE MELO MENDES

ESTE TRABALHO DE GRADUAÇÃO FOI JULGADO ADEQUADO COMO PARTE DO REQUISITO PARA OBTENÇÃO DO DIPLOMA DE
"GRADUADO(A) EM ENGENHARIA ELÉTRICA"

APROVADO EM SUA FORMA FINAL PELO CONSELHO DE CURSO DE GRADUAÇÃO EM NOME DO CURSO

Prof. Dr. Daniel Julien Barros da Silva Sampaio
Coordenador(a)

BANCA EXAMINADORA:

Prof. M.Sc. Thiago José Michelin
Orientador(a)/UNESP-FEG

Prof. Dr. Daniel Julien Barros da Silva Sampaio
UNESP-FEG

Prof. Dr. Samuel Euzédice de Lucena
UNESP-FEG

Janeiro de 2024

De modo especial, dedico esse trabalho aos meus pais, os quais sempre me deram suporte e apoio, me ensinando valores nobres os quais levarei comigo ao longo da minha vida.

AGRADECIMENTOS

Sou muito grato aos meus pais, Janira e Daniel, por todo o suporte dado ao longo da minha vida, me mostrando o valor do estudo e como a perseverança e a dedicação são capazes de gerar resultados excelentes na vida de uma pessoa, levando ao êxito acadêmico, profissional e pessoal.

Agradeço a todos os professores da universidade que passaram pela minha trajetória, contribuindo para que eu obtivesse todo o conhecimento que adquiri ao longo do curso, além de me mostrarem os princípios que formam um engenheiro e fazem o mesmo ser o profissional que é: curioso, ativo, esforçado, dedicado, solucionador, inovador, entre outros.

Agradeço meu professor orientador, Prof. M.Sc Thiago José Michelin, por me apresentar o tema desenvolvido no trabalho de conclusão de curso, me auxiliando ao longo do projeto e me motivando a aprender diversos conceitos novos.

Sou grato a todos os meus amigos, seja os de fora do ambiente da universidade, como os de dentro, me incentivando e apoiando na minha jornada ao longo do curso de graduação.

RESUMO

A automação no contexto residencial vem sendo empregada de forma crescente, visando aumento da comodidade no cotidiano dos usuários ao fornecer maior controle sobre elementos da casa. Nesse sentido, o presente trabalho busca desenvolver uma API RESTful para servir de base para um sistema de automação residencial. Como estudo de caso será realizado o controle e monitoramento de temperatura de ambientes distintos de uma casa. Para isso, será desenvolvido um servidor que realiza interface com cliente, sistema de banco de dados e dispositivos de campos. O cliente realiza requisições ao servidor visando a consulta, alteração ou adição de informações ao sistema. O sistema de banco de dados armazena informações pertinentes do sistema. Os dispositivos de campo atuam fisicamente em cada ambiente, realizando efetivamente o monitoramento e controle de temperatura. Para a comunicação apropriada entre as partes que constituem o sistema serão estudados e avaliados diferentes protocolos de comunicação, justificando seu uso para cada caso. Por fim, serão avaliados os resultados do projeto, sendo verificado se o mesmo ocorreu de acordo com o esperado.

PALAVRAS-CHAVE: API RESTful; automação residencial; protocolos de comunicação.

ABSTRACT

Automation in the residential context has been broadly used, aiming to increase convenience in users' daily lives by providing greater control over home elements. In this sense, the present project seeks to develop a RESTful API to serve as the basis for a home automation system. As a case study, temperature control and monitoring will be carried out in different environments of a house. To achieve this, a server that interfaces with clients, database system and field devices will be developed. The client makes requests to the server to query, change or add information to the system. The database system stores pertinent system information. Field devices operate phisically in each environment, effectively monitoring and controlling temperature. For appropriate communication between the parts that make up the system, different communication protocols will be studied and evaluated, justifying their use in each case. At last, the results of the project will be evaluated, checking whether it occurred as expected.

KEYWORDS: API RESTful; home automation; communication protocol.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

A automação está presente nos mais diversos âmbitos da sociedade, servindo como um recurso que facilita as atividades realizadas pelo homem. O que era previamente feito de forma braçal, e muitas vezes de forma ineficiente ou lenta, atualmente pode ser realizado de forma otimizada devido ao contínuo aprimoramento da tecnologia. O termo "automação" pode ser definido como:

> [...] qualquer sistema que utilize computação e que substitua o trabalho humano com o intuito de aumentar a velocidade e a qualidade dos processos produtivos, a segurança dos funcionários, além de obter maior controle, planejamento e flexibilidade da produção (GOEKING, 2010, p. 71).

O tópico mais pensando quando o assunto é automação é o segmento industrial, afinal, o termo surgiu no meio fabril automotivo dos Estados Unidos em 1946 (GOEKING, 2010). O conceito se expandiu para outros meios além do automobilístico, permitindo que a produção global de itens manufaturados e bens de consumo atingisse as proporções atuais, respondendo a uma demanda crescente pelos mesmos.

Em outra perspectiva, a automação ganha uma importante relevância quando se trata da facilitação da vida humana no que diz respeito a sua comodidade no segmento residencial/predial, permitindo controle e monitoramento de diversos elementos na vida cotidiana. Como apontado por Toschi (2017), a automação em setores diversificados recebe cada vez mais investimentos e atualmente está muito mais intima à vida cotidiana das pessoas, a partir do acesso de dispositivos de controle, aplicações para smartphones, etc. O processo de automação de uma casa pode ser associado ao termo "*Smart Home"*, possuindo a seguinte definição:

> [...] uma residência com tecnologia da computação e informação que antecipa e responde às necessidades dos ocupantes, trabalhando para promover seu conforto, conveniência, segurança e entretenimento através do gerenciamento da tecnologia dentro da casa e conexões com o mundo (ALDRICH, 2003, p. 17, tradução nossa).

A partir dessa crescente adoção de serviços de automação, podem ser analisados exemplos presentes na realidade das pessoas, como as assistentes virtuais Alexa e Google Home, desenvolvidos pelas empresas Amazon e Google, respectivamente. Inclusive, estudos feitos por Beirl (2019) mostraram uma análise da utilização da Alexa por 6 famílias diferentes, sendo visto que o recurso foi integrado à rotina das famílias em momentos de

lazer (jogos, dúvidas, reprodução de músicas e vídeos, entre outros), contribuindo até mesmo com o próprio aprendizado das crianças.

Ainda sobre os benefícios dos serviços de automação residencial, Wortmeyer (2005) exemplifica vantagens nos aspectos segurança e conforto. No quesito segurança, por exemplo, pode ser estabelecido um gerenciamento de diversos cômodos de uma casa a partir de sensores que identificam movimentação estranha, quando o dono está ausente. A partir desses sensores pode ser chamado automaticamente o serviço de segurança mais próximo para verificar se não houve nenhuma invasão na casa. Em relação ao aspecto conforto pode ser citada a possibilidade de controle de temperatura, iluminação e som dos cômodos da casa por meio de dispositivos eletrônicos. O usuário pode realizar esse controle dentro da casa ou até mesmo em lugares distantes.

O desenvolvimento e a abordagem de projetos, no segmento de automação residencial, são realizados de diversas formas, podendo ser moldados de acordo com as necessidades dos usuários, atendendo suas demandas específicas. A execução dos projetos deve levar em consideração a área em que será feita a automação, o propósito, os parâmetros que serão controlados, o custo de implementação, a eficiência e a robustez. Para isso, questões mais técnicas como escolha dos protocolos implementados, *hardware* (sensores, controladores, entre outros) e linguagem de programação empregada são de extrema importância.

A contextualização feita e as vantagens citadas sobre o tema automação residencial mostram o quão relevante o assunto é atualmente, fato que é justificado ainda pelo seu crescente mercado. De acordo com a Fortune Business Insights (2023), o mercado global de *smart home* foi avaliado em 2022 com o valor de USD 80.21 bilhões, com previsão de atingir o valor de USD 338 bilhões em 2030. Esses números só confirmam que o investimento no setor e uma maior exploração do tema, com a criação de projetos inovadores e desenvolvimento de novas tecnologias no segmento, são extremamente pertinentes.

1.1 OBJETIVOS

O objetivo do trabalho é o desenvolvimento de uma API RESTful para automação residencial e como exemplo de aplicação será realizado o controle e monitoramento de temperatura em ambientes distintos de uma casa. Cada ambiente da residência é dado por um cômodo que possui um circuito próprio de controle de temperatura, sendo que todas as variáveis definidas para o sistema serão registradas em uma base de dados PostgreSQL.

Será desenvolvido e implementado um Web Service para automação, que utiliza o modelo de arquitetura conhecido como API RESTful (arquitetura será referida como API REST ao longo do projeto). O sistema utilizará o modelo cliente-servidor. Nesse modelo o cliente realiza uma série de requisições ao servidor, podendo alterar o *setpoint* de temperatura de um dos ambientes controlados, obter informações atualizadas das variáveis citadas anteriormente, além de adicionar e deletar ambientes do sistema.

Cada ambiente possuirá um microcontrolador responsável pelo gerenciamento dos circuitos, com o *hardware* necessário para o controle da temperatura. Para a comunicação entre microcontroladores e o servidor será empregado um protocolo de comunicação próprio utilizando o protocolo RS-485 na camada física.

O projeto busca, portanto, desenvolver e implementar um Web Service para que diversos tipos de dispositivos clientes possam interagir com o sistema de automação. Para isso são explorados aspectos como protocolos de comunicação apropriados (HTTP, TCP-IP e RS-485), a persistência de informações através do uso de um sistema de banco de dados e a programação dos dispositivos de campo e servidor.

2 FUNDAMENTAÇÃO TEÓRICA

Para a elaboração do projeto de automação, a fundamentação teórica do funcionamento de alguns conceitos utilizados é essencial para a plena compreensão da escolha dos elementos constituintes de sua arquitetura. Os principais tópicos tratados são os protocolos de comunicação mais relevantes, assim como explicações acerca de API REST, modelo cliente-servidor e base de dados.

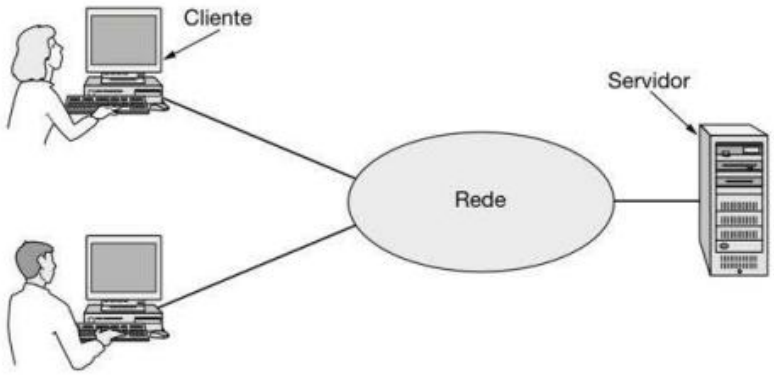
2.1 WEB E MODELO CLIENTE-SERVIDOR

A Web (*World Wide Web*) é definida por Tanenbaum (2021) como uma estrutura arquitetônica que viabiliza o acesso a documentos vinculados espalhados por diversas máquinas na Internet, tendo mudado drasticamente o cenário de compartilhamento de informações a nível mundial. Como abordado por Kurose (2021), nos primórdios da internet o acesso a mesma ocorria de forma muito limitada, sendo utilizada principalmente no meio acadêmico para serviços de correio eletrônico e transferência de arquivos. Com o início da Web (a partir da década de 90), o compartilhamento e o acesso a informações começou a ficar cada vez mais facilitado. Atualmente, por meio dessa enorme rede, qualquer usuário

com um computador conectado a Web pode acessar notícias, vídeos, imagens e trocar mensagens, sob demanda e com uma interface gráfica de alta qualidade, no momento que desejar.

No contexto da aplicação Web entra o conceito do modelo cliente-servidor, dado que a partir dele é definida a base de como é feito o acesso à rede. A Figura 1 ilustra de forma simplificada dois clientes (2 máquinas cliente rodando o processo cliente) e uma máquina servidora rodando o processo servidor.

Figura 1 - Uma rede com dois clientes e um servidor



Fonte: Tanenbaum (2021)

Na Figura 1 as máquinas cliente realizam requisições ao servidor. A máquina servidora, por sua vez, executa as rotinas necessárias para atender às solicitações do cliente e retorna uma resposta ao mesmo. Esse mecanismo expressa o funcionamento básico da relação entre cliente e servidor que permite com que toda a rede de computadores mundial funcione na Web.

Um exemplo tangível ao cotidiano é o serviço de login em um aplicativo de banco, de modo que o usuário interage com o aplicativo em seu celular (lado cliente da operação) e insere seus dados para que consiga entrar em sua conta. A partir disso, as informações inseridas são enviadas a um servidor como um tipo de solicitação, de modo que o servidor faz uma verificação dos dados e retorna o acesso para o cliente. Esse conceito se estende para muitas áreas, havendo uma série de protocolos e padrões que realizam o intermédio entre o cliente e o servidor.

2.2 PROTOCOLOS DE COMUNICAÇÃO

Como definido por Hercog (2020), o conjunto de dispositivos que se comunicam em um sistema de comunicação são chamados de entidades de comunicação. Já o protocolo de comunicação é dado pelo conjunto de regras padronizadas que rege a comunicação entre as entidades. A partir dessa definição, tem-se que os protocolos atuam de forma imprescindível no que diz respeito à organização de informações trocadas entre vários computadores em uma rede. Essa troca de dados pode ocorrer de diversas formas, surgindo cada vez mais protocolos que definem meios e regras mais eficientes e confiáveis. Como apresentado por Tanembaum (2021), existem algumas características e objetivos principais que são buscadas pelos protocolos de comunicação:

- Confiabilidade: uma informação transmitida em uma rede de computadores deve chegar ao destinatário de forma integra. É responsabilidade dos protocolos garantir que isso aconteça, utilizando para isso mecanismos de detecção e correção de erros. Esses mecanismos fazem com que a mensagem seja reenviada em caso de falhas ou que seja restaurada a partir da própria mensagem corrompida. Nesse contexto, é essencial que os dados transmitidos contenham redundância.
- Alocação de recursos: em redes com muitos dispositivos, problemas relacionados a congestionamento de dados podem ocorrer com frequência, gerando perda de informações na comunicação. Existem também questões relacionadas a fluxo de dados indevido, em que, por exemplo, um emissor envia dados de forma muito rápida a um receptor lento, o qual não consegue receber as informações na mesma velocidade. Os protocolos devem realizar uma alocação de recursos adequada, controlando o fluxo de dados e o compartilhamento de largura de banda, por exemplo.
- Capacidade de evolução: para que um sistema de comunicação/rede possa se desenvolver, aumentando o número de dispositivos pertencentes ao mesmo, é necessária uma organização adequada dos protocolos utilizados. Nesse sentido, a divisão da estrutura utilizada em camadas de protocolos, visando a fragmentação do problema maior, é uma estratégia adotada atualmente. Além disso, é pertinente o endereçamento de cada dispositivo na rede, permitindo que os emissores e receptores da comunicação sejam facilmente identificados.

• Segurança: em uma rede com diversos computadores trocando dados, evitar farsas (a partir de mecanismos de autenticação) e garantir que as informações não sejam interceptadas (confidencialidade) ou modificadas (integridade) são fatores de extrema importância.

Considerando todas os pontos citados em relação aos objetivos dos protocolos de comunicação em uma rede, foi criado pela ISO (*International Standards Organization*) o modelo OSI (*Open Systems Interconnection*). Esse modelo foi criado em 1984, possuindo o intuito de padronizar a comunicação entre dispositivos em uma rede, sendo uma referência a ser seguida.

O modelo apresenta a arquitetura considerada ideal pela organização ISO de como deve ocorrer a comunicação entre entidades distintas em uma rede. Para isso foram definidas 7 camadas diferentes, onde cada uma tem um conjunto de protocolos envolvidos, exercendo funções específicas na comunicação. Essa representação das camadas do modelo pode ser observada na Figura 2.

Figura 2 – Representação do modelo OSI e suas camadas

Fonte: Autoria própria

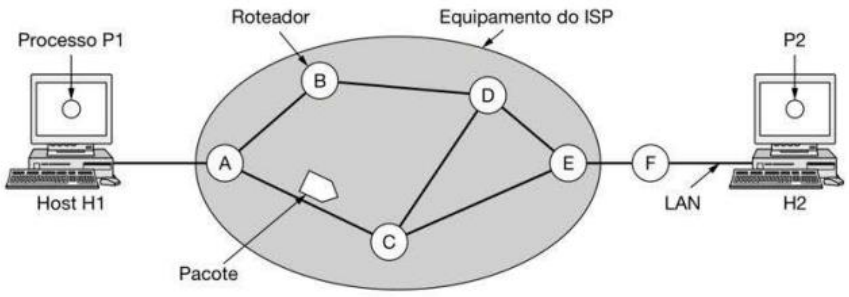
Quando uma mensagem é enviada a um determinado destinatário, diversas etapas e procedimentos com regras definidas são seguidos. Primeiramente, o usuário deve ter acesso a uma determinada interface que permita que ele comande o envio da informação. Após isso, são tratadas questões como a codificação da informação, a criptografia, a divisão da

mensagem em pacotes, a definição da rota por onde os dados são transmitidos e qual é o meio em que essa transmissão ocorre.

O modelo OSI, expresso na Figura 2, mostra idealmente cada uma das camadas que devem existir para que a comunicação seja estabelecida propriamente. No entanto, a abordagem em relação a divisão de camadas é muitas vezes tratada de forma diferente. É comum a simplificação das camadas de aplicação, apresentação e sessão em apenas uma camada de aplicação, inclusive no modo abordado por Tanenbaum (2021), o qual define as camadas da seguinte forma:

- Camada física: define as interfaces elétricas, de sincronização e outras, pelas quais os bits são enviados, para uma outra máquina, como sinais através dos canais de comunicação. É possível utilizar meios distintos para a transmissão (cabo de cobre, fibra óptica, entre outros), sendo que cada meio tem seu próprio conjunto de desvantagens em termos de frequência, largura de banda, atraso, custo e facilidade de instalação e manutenção.
- Camada de enlace: a partir dos serviços da camada física, envia e recebe bits pelos canais de comunicação, fornecendo para a camada de rede uma interface de serviço bem definida. Possui também objetivos como o enquadramento de sequência de bytes (encapsulamento de dados provenientes da camada de rede), a regulação do fluxo de dados e identificação de erros na transmissão (realizando a correção dos mesmos).
- Camada de rede: realiza à transferência de pacotes da origem para o destino, sendo que para isso são realizados “saltos” em equipamentos como roteadores, *switches* e outros dispositivos intermediários. Como pode ser visto na Figura 3, quando um host H1 envia uma mensagem para um host H2, o primeiro envia os pacotes para o roteador A, via enlace ponto a ponto, e em seguida o roteador A envia a mensagem para outro, até que a informação chegue ao host H2. Esse mecanismo é chamado de comutação de pacotes *store-and-forward*.

Figura 3 - Ambiente dos protocolos da camada de rede



Fonte: Tanenbaum (2021)

- Camada de transporte: a camada oferece um serviço confiável para o transporte de dados dos usuários (provenientes de processos da camada de aplicação). Enquanto a camada de rede oferece remessas de pacotes, a camada de transporte se baseia na de rede para proporcionar transporte dos dados de uma máquina origem para uma máquina destino. Nesse contexto, a camada de transporte atua corrigindo erros, retransmitindo dados se necessário e gerenciando a comunicação. Inclusive, caso ocorra algum problema com a conexão de rede durante a transferência de dados, a camada de transporte consegue fazer uma verificação do que já foi enviado, enviando o restante da informação quando a conexão de rede é retomada.
- Camada de aplicação: enquanto as outras camadas garantem um transporte confiável dos dados, é a camada de aplicação que está de fato mais próxima do usuário. É essa camada que contém os protocolos de nível mais alto, como o HTTP (utilizado para busca de páginas e objetos na Web) e o DNS (faz o mapeamento de nomes de hosts para seus endereços na camada de rede).

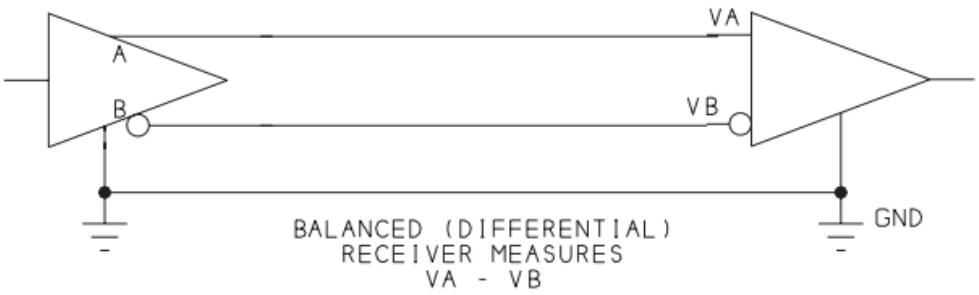
2.2.1 Protocolo RS-485

No contexto da camada física, um protocolo que ganha destaque é o RS-485 (*Recommendad Standart-485*), definido pelo padrão TIA-485-A, o qual foi criado pela *Telecommunications Industry Association*. O protocolo segue um conjunto de características para sua operação, os quais serão apresentados a seguir.

Como abordado por Axelson (2007), a comunicação no protocolo RS-485 ocorre a partir de um *driver*, o qual envia um sinal a um receptor. Esse sinal é transmitido de forma balanceada (diferencial), ou seja, a comunicação é dada por um par de fios onde cada um transmite a mesma corrente, no entanto, em sentidos opostos, o que possibilita que qualquer ruído de tensão que apareça seja cancelado ou reduzido. Em linhas desbalanceadas frequentemente ocorrem falhas de transmissão do sinal, relacionadas a existência de uma diferença de tensão entre o GND do *driver* e do receptor. Esse problema é atenuado em linhas balanceadas.

A Figura 4 apresenta a estrutura básica de uma comunicação RS-485.

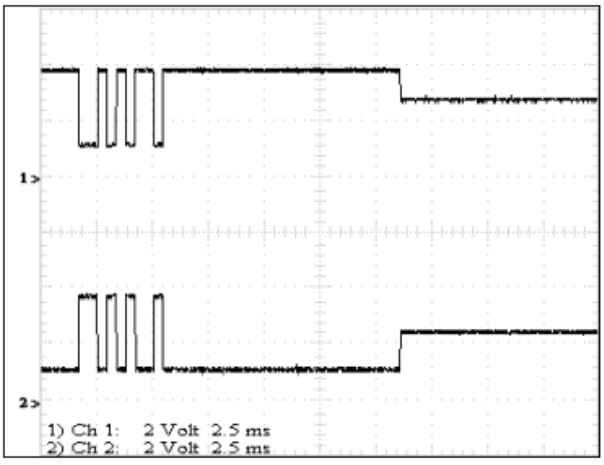
Figura 4 - Sinal balanceado (diferencial) utilizado no protocolo RS-485



Fonte: Axelson (2007)

É interessante analisar o sinal transmitido pelas linhas A e B no protocolo RS-485. Na Figura 5 podem ser observados sinais das linhas A e B de um *driver* RS-485, sendo que o bit da informação transmitida é definido pela diferença de tensão entre a e b (V_A - V_B). Um sinal HIGH é dado quando a tensão V_A - V_B é positiva, já um sinal LOW é dado quando esta tensão é negativa.

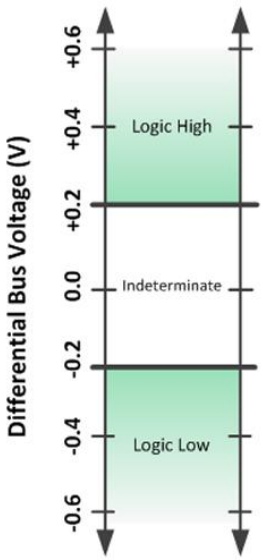
Figura 5 - Saída das linhas A e B do *driver* RS-485 como *output*



Fonte: Axelson (2007)

Como explicado por Axelson (2007), para um sinal HIGH a tensão diferencial entre A e B deve ser de no mínimo 0,2 V, enquanto para um estado LOW, a tensão deve possuir valor máximo igual a -0,2 V, como pode ser observado na Figura 6. Tem-se ainda que um *driver* só considera o sinal válido para transmissão quando $|V_{AB}| = 1,5$ V, enquanto para o receptor $|V_{AB}| = 0,2$ V é suficiente para a devida interpretação do sinal.

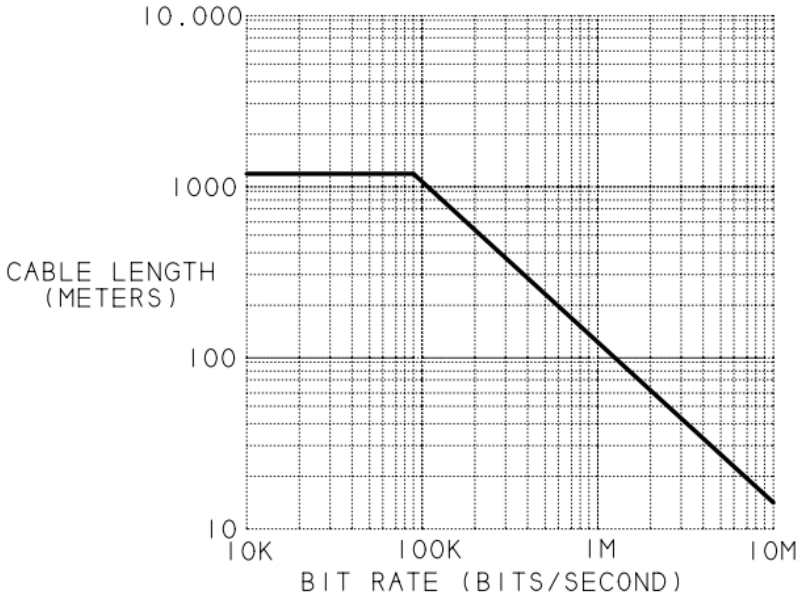
Figura 6 - Relação entre nível de tensão diferencial e nível lógico para protocolo RS-485



Fonte: Wang (2021)

A Figura 7 apresenta a relação entre taxa de transmissão em bps (bits por segundo) e comprimento do cabo. O gráfico mostra que, para curtas distâncias, é possível atingir taxas de transmissão de até 10 Mbps. À medida que a distância aumenta, a taxa de transmissão é reduzida. Este protocolo pode trabalhar com distâncias de até, aproximadamente, 1200 m sem a necessidade de repetidores.

Figura 7 - Taxa de transmissão (bits/s) do protocolo RS-485 em relação ao comprimento do cabo (m)



Fonte: Axelson (2007)

Outro ponto de importância a ser analisado na comunicação RS-485 é a necessidade de resistores de terminação da linha. Isso ocorre devido à presença de uma impedância resultante do próprio meio físico utilizado na comunicação. De acordo com Wang (2021), um barramento que utiliza cabo de par trançado como meio físico gera uma resistência na faixa de 100 Ω a 150 Ω, sendo tipicamente considerado o valor de 120 Ω.

Como explicado por Wang (2021), quando o barramento é deixado sem um resistor de terminação, ocorre uma incompatibilidade que faz com que o sinal acabe sendo refletido no final da rede, interferindo diretamente na transmissão dos bits seguintes. Se o sinal refletido estiver fora de fase em relação ao próximo bit, o nível de tensão que caracteriza esse bit pode diminuir drasticamente, gerando erros de interpretação no receptor.

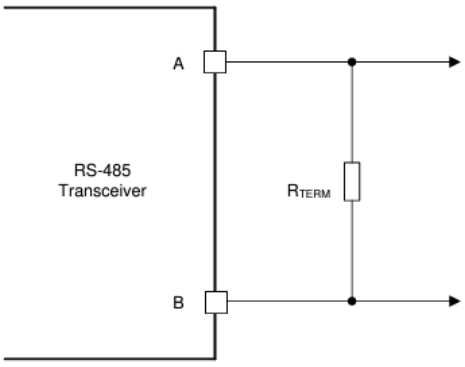
No entanto, é importante a avaliação de cada projeto para a verificação da necessidade da adição de resistores de terminação. Para isso é importante considerar principalmente dois aspectos: o tempo de propagação de um bit e o *two-way loop time* (*delay* de propagação) existentes. O primeiro parâmetro é referente ao tempo que um bit é transmitido no barramento. O segundo parâmetro se refere ao tempo que o sinal leva para sair do transmissor, chegar ao receptor e, por fim, voltar ao ponto de origem. Além disso, deve-se considerar um terceiro fator, o tempo de amostragem, que é o momento durante o período de propagação de um bit em que o receptor realiza a leitura do mesmo, interpretando se é um sinal HIGH ou LOW.

Considerando um sinal refletido devido à ausência de impedância de terminação, o mesmo vai sendo dissipado a cada reflexão sofrida, sendo diminuído o impacto na leitura dos bits seguintes gradualmente. Como evidenciado por Wang (2021), considerando 96 kΩ para impedância de entrada do barramento e uma impedância da fonte do *driver* igual a 60 Ω, tem-se que depois da sexta vez que o sinal for refletido, a magnitude do mesmo fica abaixo de 4% do valor original, não impactando mais efetivamente a leitura dos próximos bits. Essas 6 reflexões ocorrem no tempo de 3 * *two-way loop time*.

Para o cenário considerado, se o intervalo de tempo dado por 3 * *two-way loop time* for significativamente menor que o tempo em que o receptor realiza a leitura do próximo bit transmitido, não é necessária a resistência de terminação. O tempo de propagação de um bit é equivalente ao inverso da taxa de transmissão no barramento, sendo que para uma taxa igual a 9600 bps, o tempo de propagação do bit é de 104 µs. Ao considerar que o tempo de amostragem acontece na faixa de 50% a 75% desse tempo, tem-se que a amostragem ocorre no intervalo de 52 µs a 75 µs a partir do instante inicial em que o bit foi transmitido. Portanto, considerando esse exemplo, se o valor de 3 * *two-way loop time* for consideravelmente menor que o intervalo de 52 µs a 75 µs, não é necessário resistor de terminação, dado que o sinal refletido será dissipado bem antes da leitura do próximo bit.

Situações que não satisfazem o cenário descrito, necessariamente devem implementar resistores de terminação, como apresentado na Figura 8. O valor da resistência, como apresentado por Wang (2021), deve corresponder a impedância do cabo.

Figura 8 - Resistor de terminação no barramento RS-485

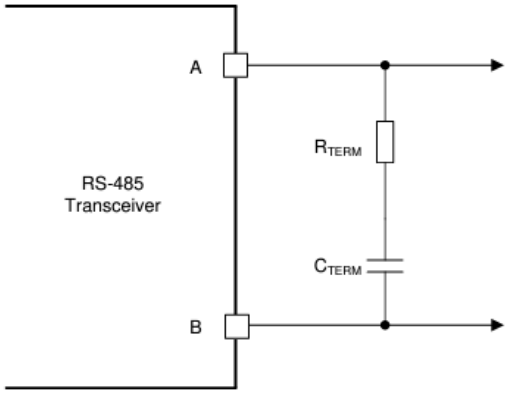


Fonte: Wang (2021)

A avaliação para a não utilização do resistor de terminação no barramento é interessante, pois essa resistência, apesar de resolver a questão da reflexão de sinais, acaba impactando o circuito negativamente em outro sentido. Como mostrado por Wang (2021), o

resistor de terminação adiciona uma carga DC no *driver*, o que aumenta a solicitação de corrente e potência do mesmo. Isso pode gerar impactos negativos na polarização do sinal transmitido. Nesse sentido, uma solução é a utilização de um capacitor em série com o resistor de terminação (configurando uma terminação AC), como apresentado na Figura 9.

Figura 9 - Terminação AC no barramento RS-485

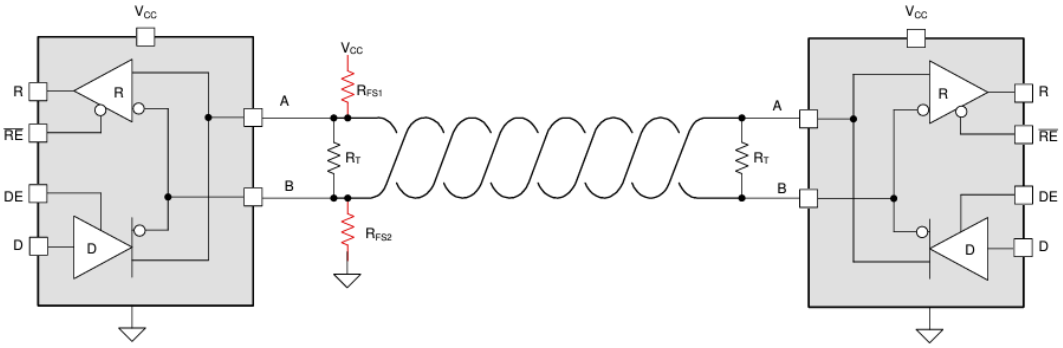


Fonte: Wang (2021)

A adição de um capacitor, como apresentado por Wang (2021), faz com que a corrente DC que passa pelo resistor de terminação seja bloqueada, resolvendo a questão do impacto dessa corrente na polarização do sinal. No entanto, o capacitor gera um *delay* capacitivo que faz com que as bordas de subida e descida dos bits transmitidos sejam mais longas, reduzindo a velocidade de transmissão dos dados.

Outro ponto importante ao tratar da comunicação RS-485 é a consideração do nível do sinal no momento em que o barramento não está sendo utilizado para transmissão de dados. Como pode ser observado na Figura 6, existe uma faixa *idle* entre -0,2 V e 0,2 V, em que o sinal está flutuando, não apresentando valor HIGH ou LOW. Como abordado por Wang (2021), visando solucionar esse estado indeterminado apresentado pelo barramento são utilizados resistores de *pull-up* e *pull-down*, representados por R_{FS1} e R_{FS2}, respectivamente, como observado na Figura 10. Esse método faz com que o barramento fique com uma tensão diferencial que estabelece o nível lógico HIGH enquanto em *idle*.

Figura 10 - Resistores de *pull-up* e *pull-down* no barramento RS-485



Fonte: Wang (2021)

As resistências R_{FS1} e R_{FS2} são conectadas, respectivamente, a uma fonte de tensão (normalmente de 5 V) e ao terra. As resistências possuem o mesmo valor para que o modo comum fique balanceado entre o terra e o V_{cc}. A disposição do circuito forma um divisor de tensão simples (considerando o resistor de terminação), que mantém a tensão diferencial no barramento com pelo menos 0,2 V.

Tratados os principais pontos sobre o funcionamento do protocolo RS-485, podem ser citadas as suas principais vantagens, de acordo com Axelson (2007):

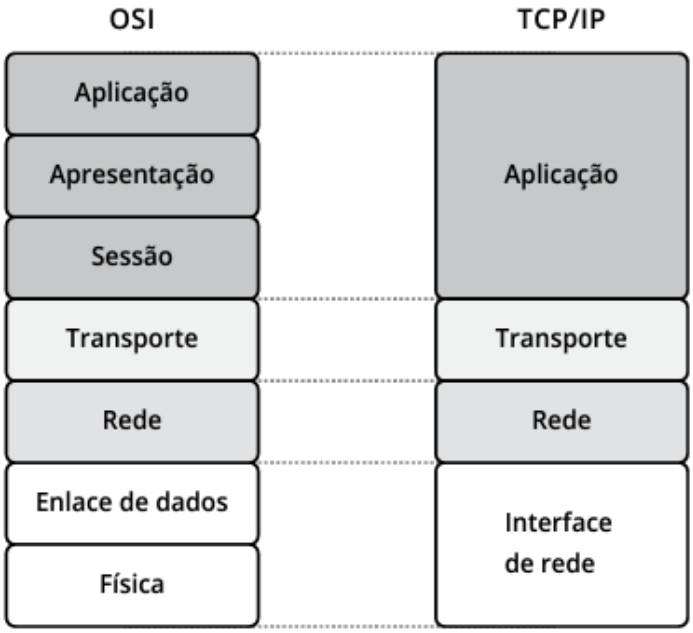
- *Drivers* e receptores são baratos, precisando apenas de uma fonte de 5 V, ou menos, para gerar a diferença mínima de 1,5 V nas saídas diferenciais;
- O protocolo não é limitado apenas a dois dispositivos, podendo atuar com topologia em barramento, o que permite a conexão de até 256 dispositivos no mesmo barramento;
- Comprimento de cabo elevado, ou seja, a comunicação pode ser feita a longas distâncias (atingindo o valor máximo de 1219,2 m);
- A velocidade do protocolo é alta, atingindo valores de até 10 Mbps.

2.2.2 TCP/IP

O TCP/IP é um conjunto de protocolos que possibilita a troca de informações entre computadores e servidores, sendo utilizado de forma majoritária nas redes existentes no mundo. O modelo TCP/IP ganhou esse nome devido ao TCP (*Transmission Control Protocol*) e ao IP (*Internet Protocol*) serem seus protocolos de destaque, sendo definido pela primeira vez em 1974 por Cerf e Kahn (TANEMBAUM, 2021).

A comparação entre o TCP/IP e o modelo OSI é frequentemente feita, dado que ambos apresentam uma estrutura semelhante e são fundamentais para o estudo de redes de computadores. Um paralelo entre o modelo OSI e o TCP/IP pode ser observado na Figura 11.

Figura 11 - Paralelo entre TCP/IP e modelo OSI



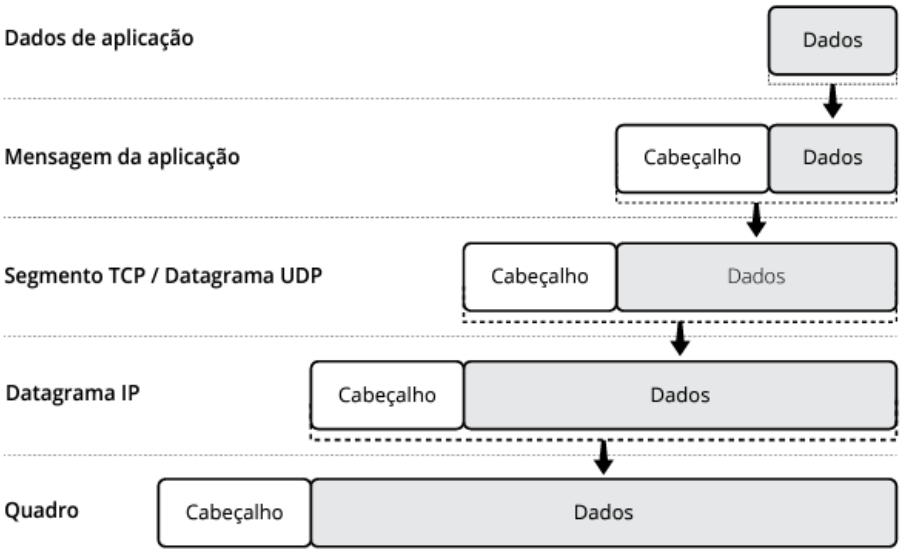
Fonte: Tanenbaum (2021)

O TCP/IP define um conjunto reduzido de camadas de protocolos quando comparado ao OSI. No desenvolvimento da arquitetura não foi vista a necessidade de definição de camadas como a de apresentação e sessão, por exemplo, podendo serem unificadas em uma única camada de aplicação. Além disso, a camada de Rede do modelo OSI seria equivalente a camada de Rede do TCP/IP, assim como as camadas de Enlace de Dados e Física correspondem à camada de Interface de Rede do TCP/IP.

Como apresentado por Elias (2013), no contexto do TCP/IP é essencial a análise de como as informações são estruturadas e repassadas entre as camadas de protocolos. A Figura 12 apresenta exatamente esse processo, sendo visto que os dados são encapsulados para a transmissão adequada entre as camadas. O encapsulamento consiste em uma camada pegar as informações provenientes da anterior e adicionar um cabeçalho, o qual possui informações de controle pertinentes para a próxima camada. Esse processo ocorre desde a camada de aplicação, onde são recebidos os dados (formando uma mensagem de aplicação com

cabeçalho), até a camada de interface de rede, em que é realizada a transmissão física dos dados ao destino. Ao chegar ao destino, é feito o processo inverso, ocorrendo o desencapsulamento das mensagens camada a camada.

Figura 12 - Encapsulamento de dados TCP/IP



Fonte: Elias (2013)

A seguir serão apresentados os protocolos TCP e IP.

2.2.2.1 Protocolo IP

O protocolo IP atua na camada de rede e é o único protocolo de rede utilizado na Internet. O IP é responsável pelo encaminhamento de informações do remetente ao destinatário via datagramas, escolhendo o melhor caminho para isso.

Como analisado por Fernández (2019), para que um sistema ofereça um serviço de comunicação universal devem ser seguidas as seguintes regras:

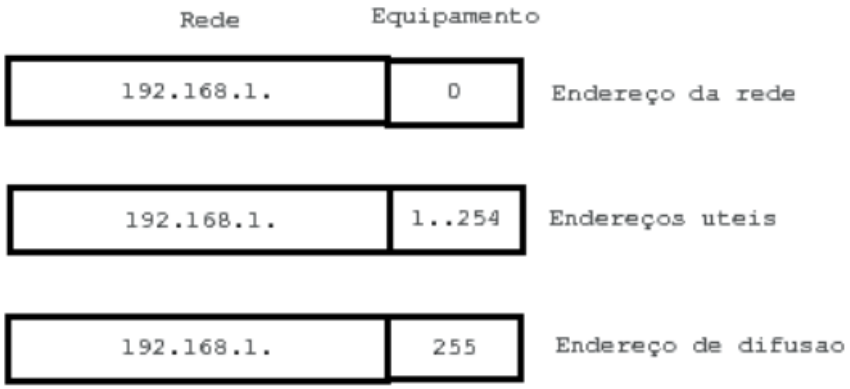
- Todos os equipamentos usam um mesmo protocolo de Rede (IP);
- Cada equipamento tem um endereço único exclusivo, isto é, só existe um determinado endereço IP no mundo.

Ou seja, o protocolo define um conjunto de endereços IP para as máquinas em uma rede, permitindo que as mesmas tenham uma identificação. É por meio desses endereços que

operam os roteadores, os quais a partir de suas tabelas de roteamento permitem que informações de um computador de uma rede A chegue a um computador de uma rede B, sendo elementos intermediários da comunicação.

O formato do endereço IP (versão IPv4) pode ser visto na Figura 13, sendo que o endereço é constituído por 4 bytes. Os 3 primeiros bytes definem a rede na qual uma máquina está inserida e o último define qual é a máquina a qual o endereço IP se refere, podendo ir de 1 a 254 (0 é destinado para identificar a rede e 255 é destinado para realizar uma transmissão de mensagens a todos os equipamentos da rede). Os endereços referentes ao IPv4 possuem classes distintas (A, B, C, D e E), cuja principal diferença é a alteração da distribuição do número de bytes destinados a definição das redes e das máquinas. A Figura 13 se refere à classe C.

Figura 13 - Esquema de endereçamento IP



Fonte: Fernández (2019)

Com o crescimento da internet, cada vez mais dispositivos foram conectados à rede, sendo observado o esgotamento dos endereços disponíveis no IPv4. Com isso, foi criado o IPv6, com capacidade para criar 1028 vezes mais endereços que o IPv4.

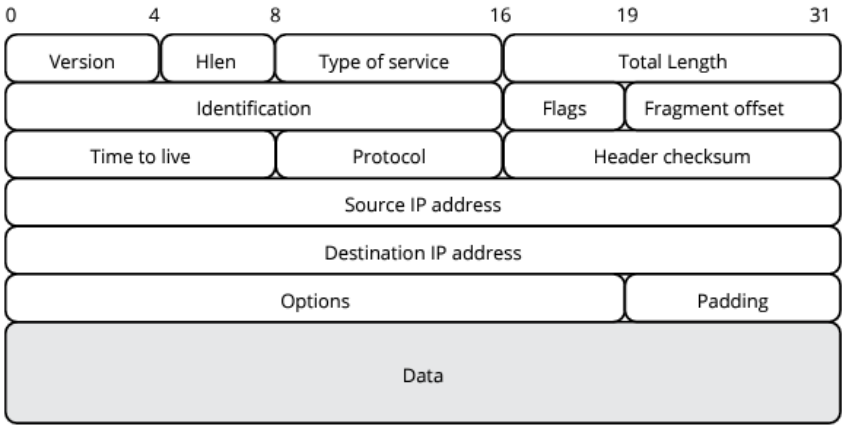
A versão IPv6 do protocolo, como apresentado por Elias (2013), define endereços formados por 8 grupos de 16 bits, separados por “:”, em que a representação de cada grupo é dada por 4 valores hexadecimais. Assim como o IPv4, é feita uma divisão dos bytes para que uma parcela identifique a rede e outra a estação. No entanto, não existem classes de endereços para o IPv6. Como exemplificado por Elias (2013), um endereço IPv6 pode ser representado pelo valor abaixo:

```
2001:0DB8:AD1F:25E2:DFA1:F0C4:5311:84C1
```

Outro ponto de destaque em relação ao funcionamento do protocolo IP é o roteamento realizado a partir de datagramas. De acordo com Elias (2013), os datagramas são a unidade básica de transferência de informações na rede, sendo estruturados de forma específica, onde cada campo carrega uma informação diferente. O protocolo IP roteia os datagramas por meio de roteadores, definindo a rota que será percorrida da origem ao destino. Além disso, são definidas regras que caracterizam aspectos como: o modo em que estações e roteadores que recebem datagramas devem processá-los, geração de mensagens de erro e quando um datagrama pode ser descartado.

Um datagrama IPv4 possui a estrutura apresentada na Figura 14, sendo notada a presença de um cabeçalho, com informações necessárias para o transporte adequado da informação ao destino. Além disso existe o campo de dados em si, referente às informações da camada de transporte, as quais são encapsuladas no datagrama.

Figura 14 - Estrutura de um datagrama IP



Fonte: Elias (2013)

A função de cada campo presente no datagrama é descrita no Quadro 1.

Quadro 1 - Campos do cabeçalho de um datagrama IP

Campo	Descrição
Source IP address	Endereço IP da estação de origem.
Destination IP address	Endereço IP da estação de destino, o qual é usado para o roteamento do datagrama.
Version	Versão do protocolo. É utilizado o número 4 para se referir ao IPv4.
Hlen	Tamanho do cabeçalho. É sempre um valor múltiplo de 32 bits (4 bytes). *Hlen* normalmente possui valor igual a 5 (20 bytes), a menos que ocorra alguma alteração definida a partir do campo *Options*.
Total length	Tamanho total (em bytes) do datagrama, incluindo tanto o cabeçalho como os dados. É um campo de 16 bits, que tem valor máximo de 65535 bytes e pode ser calculado a partir de *Total lenght* = 4 * *Hlen*.
Time to Live (TTL)	Um datagrama apresenta um número máximo de roteadores que podem fazer seu processamento, de modo que isso é controlado pelo TTL. Cada vez que o datagrama passa por um roteador, tem o valor de TTL decrementado, sendo descartado ao chegar em 0.
Protocol	Protocolo encapsulado no campo de dados, sendo que o valor 6 faz referência ao protocolo TCP, por exemplo. Quando o datagrama chega ao destino, são entregues os dados encapsulados ao protocolo indicado por *Protocol*.
Header checksum	Campo responsável por garantir a integridade do cabeçalho. No entanto, não é garantida a integridade do campo de dados.
Options	Fornece opções de teste e depuração, além de tornar variável o tamanho do cabeçalho.
Padding	Responsável por tornar o cabeçalho sempre múltiplo de 32 bits.
Data	Dados transmitidos pelo datagrama.
Type of Service	Campo de 8 bits que indica a qualidade do serviço desejado. Desses 8 bits, 3 bits são referentes a um subcampo chamado *Precedence*, que apresentam a prioridade de processamento do datagrama. Além disso, possui mais 3 subcampos (D, T e R) responsáveis pela requisição de rotas com menor retardo, maior vazão e maior confiabilidade.
Identification, Flags e Fragment offset	São campos utilizados para a fragmentação e remontagem de datagramas.

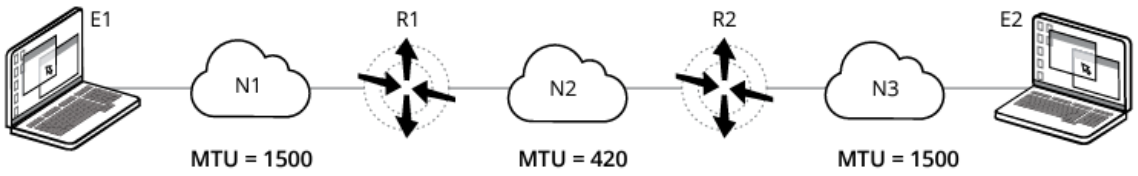
Fonte: Adaptado de Elias (2013)

Outro ponto relevante, mostrado por Elias (2013), relacionado ao roteamento dos datagramas é que os mesmos são frequentemente fragmentados pelas estações e roteadores. Isso ocorre, pois, cada rede física pela qual o datagrama é transmitido possui uma tecnologia específica que define um valor máximo para o tamanho do datagrama em MTU (*Maximum*

Transmission Unit). É nesse contexto que os campos *Identification*, *Flags e Fragmentation offset* (presentes no cabeçalho) assumem um papel relevante. Esses campos fornecem informações que permitem a remontagem adequada do datagrama ao chegar no destino, antes de seu processamento.

A Figura 15 apresenta um exemplo de fragmentação de datagramas. A rede N2 apresentação uma capacidade de transmissão máxima de datagramas de 420 bytes, enquanto as redes N1 e N3 possuem capacidade de 1500 bytes. Portanto, quando um datagrama é transmitido a partir de E1 para E2, ou vice-versa, caso o datagrama seja maior que 420 bytes, é necessário que o roteador R1 ou R2 fragmente o datagrama para possibilitar seu roteamento pela rede N2.

Figura 15 - Representação do processo de fragmentação de um datagrama IP



Fonte: Elias (2013)

2.2.2.2 Protocolo TCP

O protocolo TCP (*Transmission Control Protocol*) atua na camada de transporte e, como apresentado por Elias (2013), oferece um serviço de circuito virtual orientado a conexões.

Um serviço de circuito virtual possibilita o transporte de informações de uma aplicação a outra de forma confiável, entregando todos os dados na sequência correta, com verificação de erros. O serviço tem a denominação de ser orientado a conexões, pois antes de realizar a transferência de dados, é necessário que as aplicações envolvidas na comunicação estabeleçam uma conexão entre si. Essa conexão garante o controle de fluxo, sequência, erros e tamanho de buffers, por exemplo, sendo essencial que enquanto a transferência de dados ocorra, a conexão permaneça ativa. Quando um dispositivo A não deseja mais enviar informações a um dispositivo B, a conexão é encerrada.

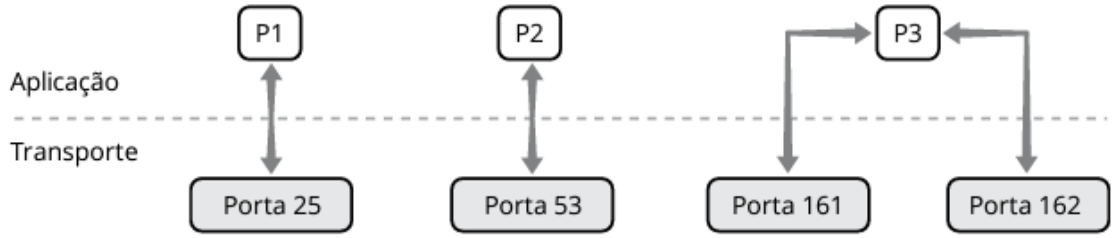
O fluxo de dados existente a partir desse circuito virtual é divido em segmentos, de modo que esses são encapsulados nos datagramas IP, ocorrendo o roteamento das informações. Os datagramas IP não possuem mecanismos que garantem que os dados

cheguem de forma completa ao destino, já que ocorre o encaminhamento de datagramas por diferentes roteadores, redes e elementos intermediários. Nesse contexto, é o TCP que atua identificando erros na transmissão, perda ou duplicação de segmentos, além de outros problemas. A partir disso são estabelecidos mecanismos para que ocorra reenvio de segmentos ou descarte, caso necessário.

Como abordado por Elias (2013), outro conceito importante ao tratar do TCP são as portas. Essa é a forma pela qual a camada de aplicação se relaciona com a camada de transporte no protocolo TCP, estabelecendo uma conexão por meio dessas para que os dados sejam recebidos e enviados.

A Figura 16 ilustra processos P1, P2 e P3 relacionados às portas 25, 53 e 161/162, respectivamente.

Figura 16 - Identificação de processos na camada de transporte



Fonte: Elias (2013)

Visando estabelecer valores únicos para a identificação de cada processo executado nas estações, é utilizado o número da porta em conjunto com o endereço IP. A identificação (191.147.09.1, 25), por exemplo, se refere a um processo que utiliza a porta 25 em conjunto com o endereço IP 191.147.09.1 para se comunicar na rede.

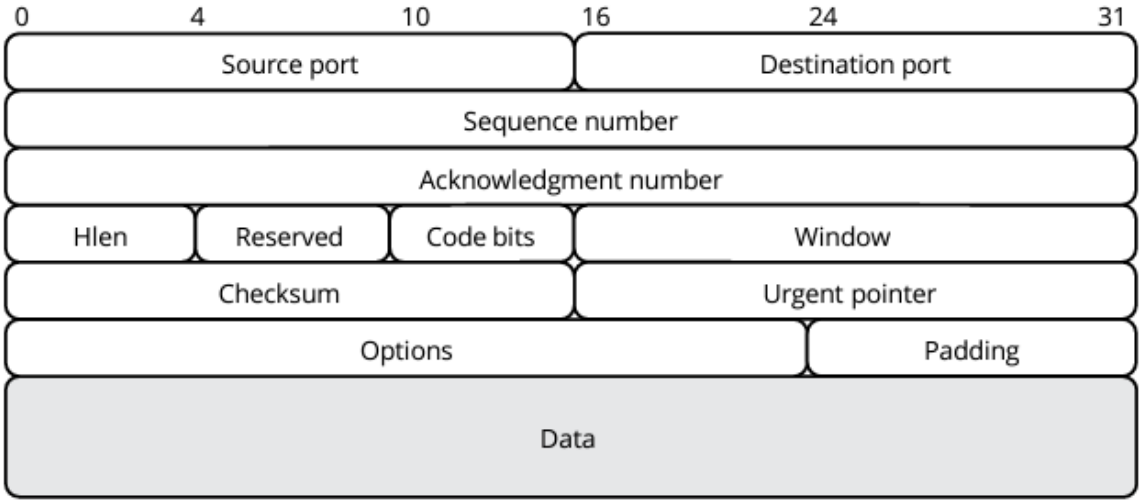
De acordo com Elias (2013), o protocolo IP guia o roteamento de datagramas (a partir do endereço IP de origem e de destino, principalmente), enquanto isso, os dados do TCP realizam a sinalização das portas de origem e de destino. Quando a informação chega ao destinatário, o TCP realiza a demultiplexação e o encaminhamento dos dados para a aplicação de destino por meio da porta, a qual identifica o ponto de recebimento dos dados. De forma análoga, o processo identificado na porta de origem permite que mensagens sejam retornadas à aplicação que enviou os dados.

De forma geral, em uma comunicação entre duas estações, a entidade TCP de origem solicita à de destino que seja aberta uma conexão, sendo definidas as portas relacionadas ao

processo, além dos recursos necessários para gerenciar o mesmo. Aberta a conexão, pode ser estabelecido um fluxo de dados entre as estações. A aplicação de origem gera mensagens que são multiplexadas em segmentos pelo TCP, que por sua vez são encapsuladas em datagramas IP, os quais são encaminhados por meio de roteadores. No destino, os segmentos transportados nos datagramas são entregues para a o protocolo TCP, sendo demultiplexados e, por fim, entregues ao processo destino. Finalizada a comunicação, pode ser encerrada a conexão, sendo solicitado seu fechamento pelas entidades TCP.

Um segmento TCP, possui a estrutura apresentada pela Figura 17.

Figura 17 - Segmento TCP



Fonte: Elias (2013)

A definição dos campos presentes no cabeçalho de um segmento são apresentados no Quadro 2.

Quadro 2 - Campos do cabeçalho do segmento TCP

Campo	Descrição
Hlen	Tamanho do cabeçalho. É sempre um valor múltiplo de 32 bits (4 bytes). *Hlen* normalmente possui valor igual a 5 (20 bytes), a menos que ocorra alguma alteração definida a partir do campo *Options*.
Checksum	Garante a integridade do segmento, verificando se ocorreu algum erro durante a comunicação. Caso ocorra um erro, o segmento é descartado, ativando o mecanismo de correção de erros.
Code bits	Campo que indica o propósito e conteúdo do segmento, sendo eles: URG (dados urgentes) , ACK (reconhecimento), PSH (mecanismo *push* adotado para encaminhar segmento), RST (conexão deve ser abortada), SYN (requisição para abertura de conexão) e FIN (requisição de fechamento de comunicação).
Options	Informações opcionais transmitidas.
Padding	Responsável por tornar o cabeçalho sempre múltiplo de 32 bits.
Data	Dados contidos no segmento.
Source Port	Identifica a entidade de comunicação de origem.
Destination Port	Identifica a entidade de comunicação de destino.
Sequence Number, Acknowledgment number, Window e Urgent pointer	Campos utilizados controle de erros, sequência e fluxo.

Fonte: Adaptado de Elias (2013)

2.2.3 Protocolo HTTP

O protocolo HTTP (*Hyper Text Transfer Protocol*) é um protocolo presente na camada mais superior dos modelos de rede abordados, a camada de aplicação. Para o entendimento das obrigações de um protocolo que atua nessa camada, Kurose (2021) lista os seguintes tópicos:

- Os tipos de mensagens trocadas (requisição e resposta, por exemplo).
- A sintaxe dos tipos de mensagens, como os campos da mensagem e como os campos são delimitados.
- A semântica dos campos (significado das informações dos campos).
- Regras para determinar quando e como um processo envia mensagens e responde a mensagens.

Os navegadores Web, como o Google Chrome e Microsoft Edge, são programas que permitem que os usuários interajam com uma série de documentos que estão alocados em um servidor. Esses documentos são denominados páginas Web, os quais possuem objetos. Os objetos por sua vez são os arquivos em si, ou seja, podem ser imagens PNG, um arquivo HTML, entre outros. Os objetos possuem URLs (*Uniform Resource Locator*), que servem como nomes mundiais para páginas, sendo esse conceito exemplificado por Kurose (2021):

```
http://www.someSchool.edu/someDepartment/picture.gif
```

No exemplo acima “http://www.someSchool.edu/” seria a parcela do URL que determina o hospedeiro do objeto que está sendo acessado, sendo que “http” se refere ao protocolo utilizado e “someSchool.edu” é o nome DNS (*Domain Name System*) do hospedeiro. Já a parcela “someDepartment/picture.gif” é o caminho do objeto na hierarquia de diretórios no servidor.

A partir do momento que o usuário insere um URL em seu navegador, de acordo com Tanenbaum (2021), os seguintes processos ocorrem:

- Navegador pergunta ao servidor de DNS qual o endereço IP do servidor;
- O DNS responde;
- O navegador estabelece uma conexão TCP para uma determinada porta do servidor;
- Navegador envia um comando HTTP solicitando a página/objeto ao servidor;
- O servidor envia a página/objeto como uma resposta HTTP;
- Se a página incluir URLs necessários para a exibição, o navegador busca esses URLs usando o mesmo processo;
- O navegador exibe a página solicitada e fecha as conexões TCP, caso não tenham mais solicitações.

Após esses passos a página solicitada a partir do URL é retornada. A Figura 18 mostra um esquemático que exibe o navegador enviando requisições HTTP para diversos servidores visando formar uma página Web com alguns objetos. Entre esses objetos podem ser citados anúncios, imagens e tabelas, cada um armazenado em um servidor diferente.

Figura 18 - A busca e a apresentação de uma página Web a partir de solicitações HTTP/HTTPS a muitos servidores

Fonte: Tanenbaum (2021)

O protocolo HTTP define métodos internos, os quais podem ser observados no Quadro 3.

Quadro 3 - Métodos internos de solicitação HTTP

Método	Descrição
GET	Lê uma página Web
HEAD	Lê um cabeçalho de uma página Web
POST	Acrescenta algo a uma página Web
PUT	Armazena uma página Web
DELETE	Remove a página Web
TRACE	Ecoa a solicitação recebida
CONNECT	Conecta através de um proxy
OPTIONS	Consulta opções para uma página

Fonte: Adaptado de Tanenbaum (2021)

Os métodos apresentados no Quadro 3 são utilizados nas solicitações feitas pelo cliente ao servidor, sendo que o tipo de método define a natureza da solicitação. Um método GET, por exemplo, é o mais comum a ser usado e indica que quem fez a requisição está solicitando que o servidor envie uma determinada informação (no contexto Web seria a solicitação de uma página). É importante ressaltar que os métodos operam de forma mais genérica, podendo se referir a objetos de forma geral. Em um sistema de monitoramento de algum parâmetro local que implementa HTTP, por exemplo, o método GET pode ser usado justamente para

obter um parâmetro em específico e não uma página com diversos objetos, como no contexto Web.

Como resposta de uma solicitação feita por um cliente, um servidor pode retornar um conjunto de códigos de *status*, como pode ser visto no Quadro 4. Esses códigos são mensagens de respostas enviadas pelo servidor, indicando o resultado da solicitação.

Quadro 4 - Os grupos de respostas de código de *status*

Código	Significado	Exemplos
1xx	Informação	100 = servidor concorda em tratar da solicitação do cliente
2xx	Sucesso	200 = solicitação com sucesso; 204 = nenhum conteúdo presente
3xx	Redirecionamento	301 = página movida; 304 = página em cache ainda válida
4xx	Erro do cliente	403 = página proibida; 404 = página não localizada
5xx	Erro do servidor	500 = erro interno do servidor; 503 = tente novamente mais tarde

Fonte: Adaptado de Tanenbaum (2021)

Como observado no Quadro 4, cada valor de código possui um significado em relação a operação, seja uma falha do cliente (código 4xx), seja o sucesso da solicitação (200), tornando a sua existência no corpo da mensagem HTTP muito relevante.

2.3 WEB SERVICES

Como apresentado por Automation Step by Step (2016), para o entendimento do que são Web Services pode ser realizada uma analogia. Em um restaurante existe um cliente que fala apenas o idioma português e um chefe de cozinha que fala apenas inglês. De alguma forma, deve ser realizada uma comunicação entre ambos que permita que seja feito o pedido da refeição. Nesse caso, a comunicação pode ser intermediada por um garçom que fala ambas as línguas, podendo interagir com o cliente e com o chefe de cozinha.

A situação descrita pode ser associada ao contexto de Web Services, em que o chefe de cozinha seria análogo a um servidor, o cliente do restaurante a um cliente e o garçom ao Web Service. O Web Service, portanto, é uma serviço cuja principal função é permitir a integração entre diferentes aplicações, sendo realizada a troca de informações em um formato que seja compreensível para ambas. Um Web Service é uma API (*Application Programming Interface*), sendo que a diferença é que o primeiro atua no contexto da Web, enquanto o segundo não necessariamente.

Atualmente os Web Services são utilizados em diversos contextos. Uma primeira aplicação (desenvolvida em Python e integrada com uma base de dados MySQL) e uma segunda (desenvolvida em Go e integrada com Oracle) podem se comunicar por meio de um Web Service, por exemplo. Além disso, a ferramenta permite que desenvolvedores integrem seus projetos a serviços de terceiros. Um exemplo seria um aplicativo que necessita de acesso ao recurso de mapas, sendo possível para isso a utilização de Web Services disponíveis no sistema operacional utilizado no projeto. Ou seja, já existe uma interface que fornece o serviço ao desenvolvedor, permitindo que seu aplicativo e o sistema de mapas troquem informações.

Os Web Services são explicados a partir de um cliente (consumidor do serviço) e de um servidor (provedor de serviço). Nesse contexto, o primeiro realiza requisições ao segundo, o qual por sua vez retorna uma resposta (processo ocorre de acordo com o protocolo HTTP). Além disso, são utilizados o XML (*Extensible Markup Language*) ou o JSON (*JavaScript Object Notation*), que são formatos de dados que possibilitam a transferência de informações entre ambas as partes. As Figuras 19 e 20 apresentam exemplos das estruturas de dados destes formatos. Nesses exemplos são representados dados a respeito de uma loja, sendo informado o id, o C.E.P, o nome e o número de funcionários da loja.

Figura 19 - Exemplo de informações no formato JSON

```
{
    "id_loja": 123,
    "cep": 12345678,
    "nome": "abcde",
    "numero_funcionarios": 10
}
```

Fonte: Autoria própria

Figura 20 - Exemplo de informações no formato XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
    <id_loja>123</id_loja>
    <cep>12345678</cep>
    <nome>abcde</nome>
    <numero_funcionarios>10</numero_funcionarios>
</root>
```

Fonte: Autoria própria

Em relação aos tipos de Web Services, os dois mais implementados atualmente são o SOAP (*Simple Object Access Protocol*) e o REST (*Representational State Transfer*).

O SOAP, como apresentado por IBM (2021), é um protocolo que necessariamente usa como base o formato XML, utilizando também o protocolo HTTP na maioria dos casos. Os padrões do SOAP são definidos a partir da W3C (*World Wide Web Consortium*).

Uma mensagem SOAP envolve um envelope no qual é feito a definição de uma estrutura que descreve o que está na mensagem e como a mesma deve ser processada (IBM, 2021). Mais especificamente, contém informações de controle, o endereço e a mensagem que será transmitida em si. Já a mensagem, por sua vez, contém seu corpo (identificação da mensagem e seus parâmetros) e zero ou múltiplos cabeçalhos (responsáveis por transportar informações de controle, como autenticação e dados de roteamento).

O outro tipo de Web Service mais utilizado é o REST, sendo especificado a seguir.

2.3.1 REST

Como um Web Service é uma API, é comum se referir a um Web Service que aplica o padrão REST como API RESTful, ou de forma simplificada API REST.

O REST não é um protocolo, mas sim um estilo de arquitetura e um conjunto de restrições que podem ser aplicados a uma API.

Segundo Automation Step by Step (2016), as principais características do padrão REST são definidas a seguir:

A primeira característica é sua interface uniforme. Para uma API REST diversos elementos de uma aplicação podem ser caracterizados como recursos (documentos, imagens, entre outros). Por exemplo, em uma aplicação pode ser definido um tipo “Funcionario”, o qual contém os atributos “ID”, “Nome” e “Idade”. Nesse contexto, o padrão REST permite que sejam criados recursos associados a cada um dos parâmetros descritos, de modo que os identificadores desses recursos são chamados de URI (*Uniform Resource Identifier*). Esses recursos podem ser transferidos entre cliente e servidor quando é definida uma aplicação com um determinado domínio.

Considerando o domínio “teste”, por exemplo, poderia ser acessado o recurso “funcionario” a partir da seguinte rota:

```
http://teste.com/funcionario
```

Caso fosse desejado o acesso a um funcionário específico poderia ser utilizada a seguinte rota:

```
http://teste.com/funcionario/{id}
```

Na rota acima, {id} deve ser substituído pelo valor numérico identificador de um funcionário específico.

Para a realização de manipulações dos recursos, o padrão REST utiliza os métodos HTTP (apresentados no Quadro 3). Com o método POST, por exemplo, poderia ser feita uma requisição ao servidor para adicionar um novo funcionário ao sistema. Por outro lado, utilizando o método GET poderiam ser obtidos os dados de diversos funcionários ou de um em específico (ao identificá-lo pelo "id").

Outro ponto em relação aos recursos é que os mesmos podem possuir diferentes tipos de representação (como JSON, XML e HTML), o que é definido a partir da relação entre o cliente e o servidor.

A segunda característica da API REST é o fato de ser *stateless.* Como definido por Step by Step (2016), toda requisição estabelecida deve ser independente uma da outra. Uma requisição feita por um cliente deve conter todos os dados para que o servidor consiga processá-la adequadamente, sendo, portanto, autossuficiente.

A terceira característica é em relação a performance da API REST, devendo ser citada a utilização de armazenamento em cache. Quando é feita uma requisição ao servidor, é retornado ao cliente a resposta em si, mas também é retornado no cabeçalho informações sobre a necessidade de armazenamento da resposta a nível local e por quanto tempo deve permanecer armazenada. Isso impacta positivamente a performance da API, evitando a realização de operações repetidas entre o cliente e o servidor.

Por fim, a quarta característica destacada é a flexibilidade de uma API REST, dado que as partes que compõem o servidor são altamente desacopladas, permitindo que cada uma possa evoluir de forma separada (não é impactado o desenvolvimento de outras partes, nem mesmo na aplicação do cliente).

2.4 BASE DE DADOS

Atualmente, o grande volume de informações em diversos âmbitos da sociedade fazem com que o registro e a organização de informações sejam imprescindíveis. Uma empresa que contém unidades de produtos distintos em estoque, por exemplo, deve de alguma forma manter controle do que entra e sai do estoque, além de quantas unidades existem de um determinado produto (visando a avaliação da necessidade de emissão de uma ordem de compra para atender a demanda). Nesse contexto, a base de dados pode permitir, dependendo de sua configuração, uma grande rastreabilidade das movimentações, acessos e modificações dos dados armazenados. Com isso, caso ocorra algum problema de recebimento de produto para abastecimento de estoque, por exemplo, pode ser verificada a sequência de registros que envolvem desde a ordem de compra, até possíveis cancelamentos realizados.

Como definido por Amadeu (2015), um banco de dados é uma coleção de dados relacionados, os quais por sua vez são fatos conhecidos que podem ser registrados e possuem significado implícito, como nomes e números de telefone. Uma base de dados deve se atentar aos seguintes pontos (MEDEIROS, 2013):

- Integração: todos os usuários que acessam a base de dados devem ter acesso às mesmas informações;
- Inconsistência: dados ficam muito tempo sem atualização, sendo fornecidos dados antigos e muitas vezes antiquados aos usuários;
- Redundância: inclusão de informações repetidas na base de dados, o que gera uma ineficiência que gera maior gasto de armazenamento.

Além disso, uma base de dados deve possuir as seguintes características (AMADEU, 2015):

- Representação de algum aspecto do mundo real;
- Possuir finalidade específica;
- Possuir grupo definido de usuários;
- Possuir aplicações previamente concebidas, passível de interesse dos usuários;
- Ser coleção coerente de dados com significado inerente e não aleatório.

Apresentadas as principais características de uma base de dados, é de grande relevância a introdução da SQL (*Structured Query Language*), que é uma linguagem utilizada em base de dados relacionais, caracterizada pelo tratamento dos dados como relações e conceitos matemáticos (armazenamento em tabelas). Como apresentado por Alves (2014), a partir do SQL é possível realizar diversas operações em um banco de dados, seja remoção, consulta ou inclusão de dados. No entanto, SQL não é uma linguagem de programação por si só, apesar de possuir integração com diversas linguagens (como Go, C/C++, entre outras), as quais permitem a execução de *querys*.

Os comandos em SQL podem ser divididos principalmente em DDL (*Data Definition Language*) ou DML (*Data Manipulation Language*). O primeiro é responsável pela criação, alteração e remoção de banco de dados, tabelas, entre outros, já o segundo é utilizado para alteração e manipulação de dados em si. O Quadro 5 apresenta os principais comandos DDL, enquanto o Quadro 6 apresenta os principais DML.

Quadro 5 - Comandos de definição de dados (DDL)

Comando	Definição
CREATE DATABASE	Criar base de dados vazia
ALTER DATABASE	Permitir alterações em algumas características da base de dados
DROP DATABASE	Apagar um banco de dados existente
CREATE TABLE	Criar uma tabela de dados
ALTER TABLE	Permitir alterações na estrutura de uma tabela existente
DROP TABLE	Apagar uma tabela de dados existente
CREATE INDEX	Criar índices secundários para uma tabela
DROP INDEX	Apagar um índice existente
CREATE DOMAIN	Criar um domínio para campos das tabelas
DROP DOMAIN	Apagar um domínio existente
CREATE VIEW	Criar uma visão com base em uma ou mais tabelas
DROP VIEW	Apagar uma visão existente

Fonte: Adaptado de Alves (2014)

Quadro 6 - Comandos de manipulação de registros (DML)

Comando	Função
INSERT INTO	Inserir um novo registro na tabela de dados
DELETE FROM	Apagar um ou mais registros de uma tabela de dados
UPDATE	Permitir que os dados de um registro sejam atualizados
SELECT FROM	Selecionar um conjunto de registros a partir de uma condição e retorná-los ao usuário

Fonte: Adaptado de Alves (2014)

3 SOFTWARES E FERRAMENTAS UTILIZADAS

Para o desenvolvimento e teste do projeto foram utilizados alguns *softwares*, sendo pertinente introduzi-los. Os principais programas utilizados foram: Proteus 8, PostgreSQL, pgAdmin 4 e Postman. Além disso, foi utilizada a linguagem Go para a programação do servidor.

3.1 PROTEUS 8

O *software* Proteus é uma aplicação voltada ao desenvolvimento e simulação de circuitos eletrônicos. Através desta ferramenta, é possível elaborar esquemáticos e gerar arquivos para posterior construção de placas de circuito impresso. O programa fornece uma interface amigável que permite a adição de elementos ativos (como fontes de tensão e de corrente), elementos passivos (resistores, capacitores, indutores, entre outros), semicondutores (diodos e transistores, por exemplo), além de uma grande quantidade de circuitos integrados amplamente utilizados na eletrônica. A plataforma conta também com bibliotecas online que permitem a adição de ainda mais componentes para teste.

3.2 POSTGRESQL E PGADMIN 4

O PostgreSQL, como abordado por Juba (2015), é um Sistema Gerenciador de Banco de Dados Relacional (SGBDR) *open-source* e gratuito. O PostgreSQL tem seu nome com origem em 1977, quando Mikael Stonebraker criou o projeto Ingres, um SGBDR baseado no modelo relacional formal. O PostgreSQL assumiu esse nome em 1996, com sua primeira versão *open-source* lançada em 1997. Atualmente é utilizado a nível mundial nas mais diversas aplicações, como exemplo, podem ser citados: o Skype, o Instagram e a *American Chemical Society*.

O PostgreSQL possui uma série de características relevantes (JUBA, 2015):

- *Open-source:* o SGBDR possui código fonte aberto e uma grande comunidade de desenvolvedores ativa, cooperando para atualizar a documentação e realizar o lançamento de novas versões constantemente. A comunidade possui até mesmo um serviço agregador de blogs, chamado *Planet PostgreSQL*, o qual é utilizado por empresas e desenvolvedores para o compartilhamento de experiências;

- Escalabilidade: possui grande capacidade de crescimento sem queda na performance;
- *Cross-platform*: opera em diferentes ambientes e sistemas operacionais, possuindo *drivers* para as mais modernas linguagens de programação;
- Segurança: suporta diversos métodos de autenticação, como Kerberos, GSSAPI e RADIUS, além de poder utilizar formas de criptografia;
- Suporte a diversos tipos de dados: possui uma rede central de distribuição (PGXN) que permite aos usuários baixarem extensões que possibilitam acesso a novos tipos de dados. Entre os dados suportados pelo PostgreSQL podem ser citados dados geométricos, endereços de rede, *Universal Unique Identifiers*, XML, JSON, Hstore, entre outros.

Os fatores citados evidenciam a qualidade e o propósito do SGBDR, o que justifica seu uso no presente projeto para o armazenamento de informações.

O *software* pgAdmin 4 é uma aplicação que atua de forma conjunta à base de dados PostgreSQL. Fornece uma interface gráfica amigável e possui funcionalidades como administração de servidores, monitoramento, consulta de dados, diagnóstico e desenvolvimento de diagramas para base dados.

3.3 POSTMAN

Postman é uma plataforma dedicada ao desenvolvimento de APIs, permitindo a criação e gerenciamento de diferentes requisições HTTP, as quais podem ser devidamente organizadas em diferentes áreas para aplicações diversas. A aplicação também apresenta uma interface amigável, facilitando a manipulação do corpo das requisições, por exemplo, já discriminando o formato desejado (JSON, Javascript, HTML, entre outros). Tendo em vista essas características, a realização de testes e a organização das requisições na plataforma foi de grande utilidade para desenvolvimento do projeto.

3.4 LINGUAGEM DE PROGRAMAÇÃO GO

Como apresentado por Fireship (2021), a linguagem de programação Go foi criada pela empresa Google em 2007, possuindo como um de seus desenvolvedores Ken Thompsom, conhecido pela criação da linguagem B (a qual, por sua vez, serviu como base para a linguagem C). A primeira versão *open-source* de Go foi lançada em 2009, sendo uma

linguagem que atualmente é utilizada por diversas empresas, como Google, Mercado Livre, Microsoft e Netflix.

Segundo Fireship (2021), a linguagem Go apresenta diversas características importantes no contexto deste projeto. A primeira é sua performance, a qual é melhor que linguagens interpretadas (como Python e JavaScript). Em relação à compilação da linguagem, essa é muito veloz, sendo feita para código de máquina. Ao ser realizada a compilação é gerado um arquivo executável que pode ser executado em qualquer computador ou sistema operacional, não sendo necessário um ambiente de execução (*runtime environment*) próprio da linguagem para isso.

Outro ponto notável é a sua simplicidade, possuindo uma sintaxe concisa e de fácil entendimento, apresentando também *type inference* (capacidade de serem deduzidos os tipos das variáveis com base em seu contexto). Além disso, possui um sistema de módulos e pacotes que torna fácil a importação e exportação de código. Como explicado por Código Fonte TV (2019), a necessidade de uma linguagem de fácil utilização foi uma das principais razões que motivou o Google a desenvolver o Go. No contexto da época, as linguagens existentes eram muitas vezes consideradas complexas, sendo que Go surgiu como uma opção que além de simples, tornou os processos do Google mais produtivos e escaláveis.

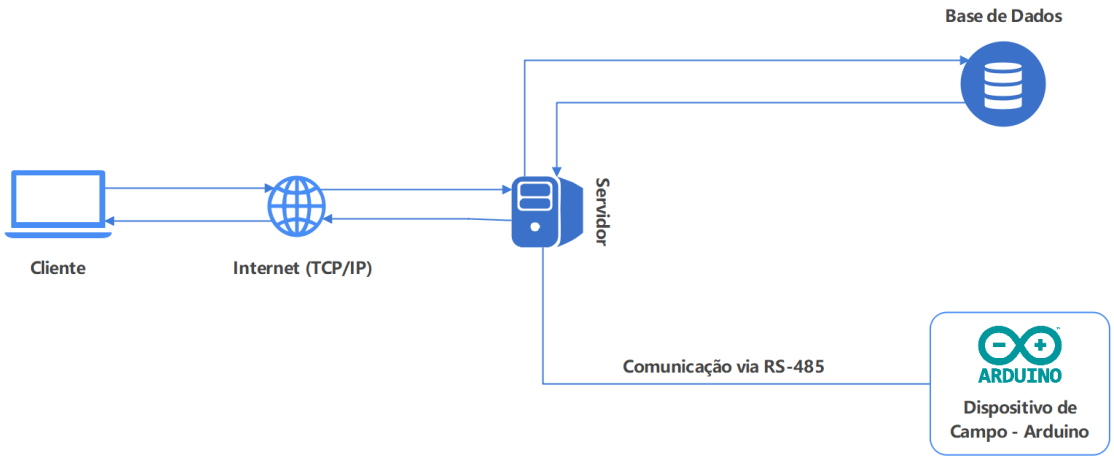
As características citadas justificam a excelência da linguagem para aplicações Web, motivo pelo qual foi escolhida para o desenvolvimento do servidor do presente projeto. Como abordado por Go (2019), a linguagem suporta as últimas tecnologias HTTP, base de dados como MySQL, PostgreSQL e MongoDB, além de padrões de criptografia modernos, como o TLS 1.3.

4 DESENVOLVIMENTO

Nesse trabalho será realizado o desenvolvimento de uma API REST. O objetivo é criar uma ferramenta de *software* para automação de aplicações residenciais, que permita o monitoramento e controle de diversos dispositivos que compõem tal sistema. Ao mesmo tempo, esta ferramenta proporciona uma forma de acesso simples para que usuários possam interagir com o sistema utilizando dispositivos conectados à rede (celular, notebook, por exemplo). Como estudo de caso, será implementado um sistema de controle de temperatura para os ambientes onde o sistema está inserido.

A Figura 21 apresenta um diagrama representativo de todas as partes constituintes do projeto.

Figura 21 - Estrutura geral do projeto



Fonte: Autoria Própria

O servidor figura como elemento central do sistema, realizando interface com o cliente, com a base de dados e com os dispositivos de campo. Adiante serão introduzidas cada parte do sistema.

Os dispositivos de campo/microcontroladores são elementos que ficarão dispostos fisicamente em ambientes/cômodos de uma casa. Cada ambiente apresenta também dispositivos de refrigeração/aquecimento e sensores de temperatura, os quais são gerenciados e comandados pelos microcontroladores. Com base nisso os dispositivos de campo conseguem regular a temperatura do ambiente em torno de um valor específico, dado por um *setpoint* fornecido pelo usuário.

Por meio de uma conexão via RS-485 (topologia em barramento) com o servidor, o microcontrolador consegue trocar dados com o mesmo. A comunicação envolve o fornecimento de dados de temperatura para o servidor e o envio do valor de *setpoint* para o microcontrolador.

Em relação à base de dados, a mesma possuirá a função de armazenar informações características dos ambientes monitorados, assim como dos dispositivos de aquecimento e refrigeração. O servidor será responsável por manipular de forma adequada esses dados (exclusão ou adição de ambientes, alteração ou resgate de parâmetros).

No que tange a comunicação entre clientes e servidor são estabelecidas rotas, as quais são endereços utilizados para que um usuário da API possa manipular recursos do servidor. As rotas são apresentadas no Quadro 7, o qual informa o tipo de requisição, terminal da rota utilizada e função da requisição.

Quadro 7 - Rotas implementadas para o projeto

Método	Rota	Função
`GET`	`/DadosAmbientes`	Obter informações de todos os ambientes
`GET`	`/DadosAmbientes/{ambienteID}`	Obter informações de um ambiente específico, a partir de um ID que o identifica
`POST`	`/AdicionarAmbiente`	Cadastrar um novo ambiente, para monitoramento e controle, no sistema
`DELETE`	`/DeletarAmbiente/{ambienteID}`	Deletar um dos ambientes cadastrados do sistema, a partir de um ID que o identifica
`PATCH`	`/AtualizarSetpoint/{ambienteID}`	Atualização de setpoint de temperatura de um ambiente em específico, a partir de um ID que o identifica
`GET`	`/DadosAmbientes/DadosDevices`	Obter informações acerca dos dispositivos de refrigeração/aquecimento presentes em cada ambiente

Fonte: Autoria Própria

Antes dos clientes realizarem requisições ao servidor, o mesmo deve ser configurado de forma apropriada. Nesse sentido, o protocolo TCP estabelece uma conexão segura e confiável entre ambas as partes, sendo necessário para isso a definição da porta por onde essa conexão ocorrerá. No caso foi utilizada a 8080. Definida a porta, o servidor é inicializado de fato, começando a “escutar”, aguardando por conexões. Além disso, são feitas verificações de erros de modo a assegurar que o servidor foi inicializado corretamente. Esse procedimento descrito

pode ser observado das linhas 11 a 23 do código apresentado na Figura 22. O trecho de código foi retirado do arquivo main.go, o qual pode ser visto no Apêndice A.

Figura 22 - Inicialização do servidor

```
1
2  router := chi.NewRouter()
3
4  go getAllTemps(port, db, channelOne, channelTwo)
5  router.Get("/DadosAmbientes", handlerGetAllAmb(db))
6  router.Get("/DadosAmbientes/{ambienteID}", handlerGetSpecificAmb(db))
7  router.Get("/DadosAmbientes/DadosDevices", handlerGetAllDevices(db))
8  router.Post("/AdicionarAmbiente", handlerPostAmb(db, port, channelOne, channelTwo))
9  router.Delete("/DeletarAmbiente/{ambienteID}", handlerDeletAmb(db))
10 router.Patch("/AtualizarSetpoint/{ambienteID}", handlerUpdateSetpoint(db, port, channelOne, channelTwo))
11
12 server := &http.Server{
13     Handler: router,
14     Addr:    ":8080",
15 }
16
17 log.Printf("Server starting on port 8080")
18
19 err = server.ListenAndServe()
20
21 if err != nil {
22     log.Fatal(err)
23 }
```

Fonte: Autoria Própria

Na Figura 22 também pode ser observada a definição de um objeto “router” na linha 2, que é uma instância de um roteador definido a partir da biblioteca “chi” da linguagem Go. Esse “router” associa as rotas às funções *handlers* responsáveis por tratar dados das requisições, permitindo que seja retornada uma resposta adequada ao usuário. Na linha 14 é associado “router” à interface “Handler” do servidor.

O código da linha 4 a 10, da Figura 22, apresenta como são tratadas todas as requisições. Na linha 5, por exemplo, é definido um método “router.Get()” (utilizado para tratar uma requisição do tipo GET). O método recebe como parâmetros a rota “/DadosAmbientes” e uma função “handlerGetAllAmb(db)”. O primeiro parâmetro se refere a rota cuja requisição associada será tratada, já o segundo se refere a uma função criada pelo desenvolvedor, responsável por processar os dados da requisição e retornar as informações de todos os ambientes cadastrados, nesse caso. Essa função também recebe como parâmetro uma variável “db”, referente à conexão com o banco de dados, para que a função *handler* consiga executar *queries* na base de dados.

Apresentado o panorama geral do projeto, serão detalhadas todas as partes que compõem o sistema, sendo essas: a modelagem da base de dados, o funcionamento da

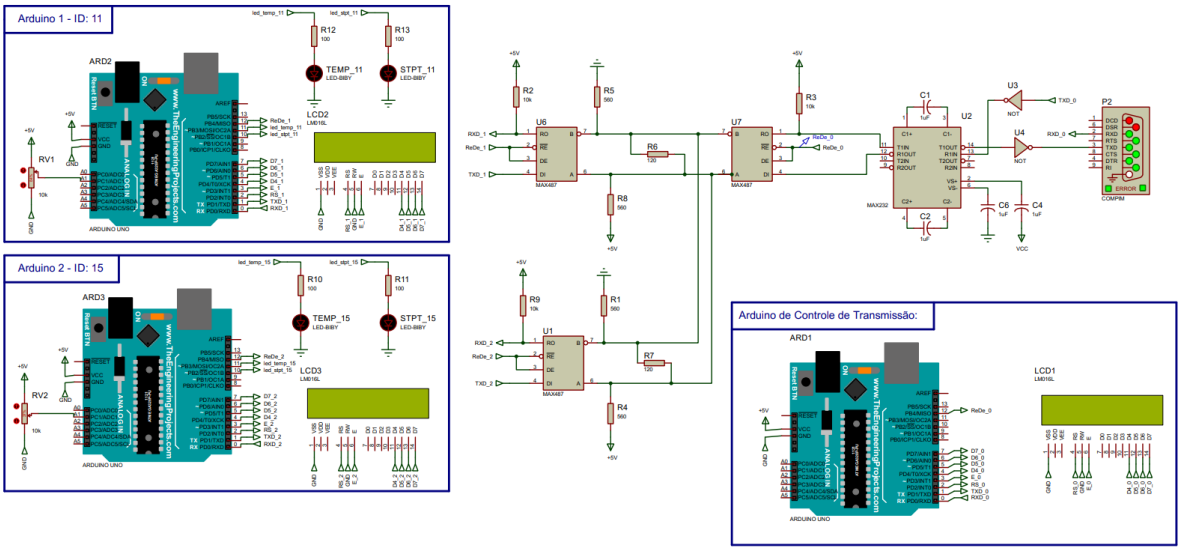
comunicação mestre-escravo entre o servidor e os dispositivos de campo, a definição da estrutura das requisições e operações desenvolvidas, além da verificação de erros.

O código desenvolvido para o servidor pode ser observado nos Apêndices A, B, C, D, E, F, G, H, I, J, K e L. O código desenvolvido para os microcontroladores, por sua vez, pode ser observado no Apêndice M.

4.1 CIRCUITO COM DISPOSITIVOS DE CAMPO

A partir do *software* Proteus 8 foi criada e simulada a parte física do projeto, o que inclui os microcontroladores (foram utilizados arduinos uno) e a comunicação estabelecida entre os microcontroladores e o servidor. A Figura 23 mostra o circuito desenvolvido em sua totalidade.

Figura 23 - Circuito contendo dispositivos de campo e a estrutura da comunicação física de dados

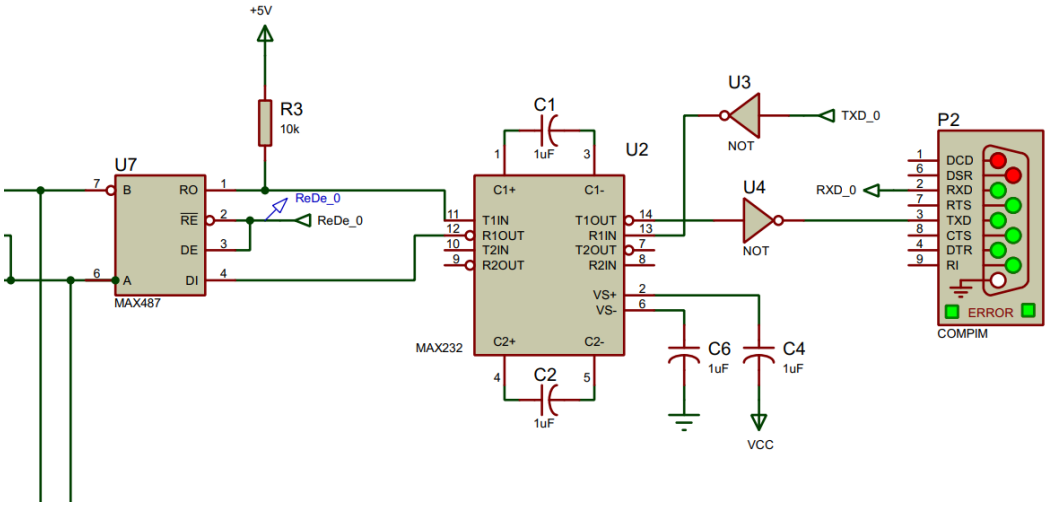


Fonte: Autoria Própria

Como pode ser observado, para o estabelecimento da comunicação RS-485 entre o servidor e o arduino foram utilizados alguns componentes integrados, visando a realização da compatibilidade dos sinais. Cada trecho do circuito será apresentado a seguir.

A Figura 24 mostra o trecho inicial do circuito, contendo uma porta COM à direita, seguido por 2 componentes integrados, o MAX-232 e o MAX-487.

Figura 24 - Trecho do circuito referente à interface entre servidor (porta serial) e linha RS-485



Fonte: Autoria Própria

Como abordado por Axelson (2007), o termo “porta serial” se refere a portas que utilizam um protocolo assíncrono, incluindo as portas RS-232 nos computadores e portas seriais em sistemas embarcados. A transmissão dos dados ocorre, na maioria das vezes, de forma bidirecional, sendo enviado um bit de cada vez. Nos computadores, aplicações acessam a maior parte das portas seriais por meio de portas COM.

Axelson (2007) destaca algumas vantagens das portas seriais:

- Podem transmitir qualquer tipo de dado;
- O *hardware* necessário é muito barato e disponível no mercado (maioria dos computadores e microcontroladores possuem a porta serial). Cabos utilizados na comunicação também são baratos;
- Sistemas operacionais, como o Windows, fornecem *drivers* para o acesso a portas COM. Além disso, diversas linguagens de programação possuem repositórios e outras ferramentas que permitem o acesso a essas portas COM;

As características citadas mostram a praticidade, baixo-custo, popularidade e facilidade de acesso e controle das portas seriais. Esses fatores tornam a porta serial adequada para a realização da interface entre o servidor e os microcontroladores.

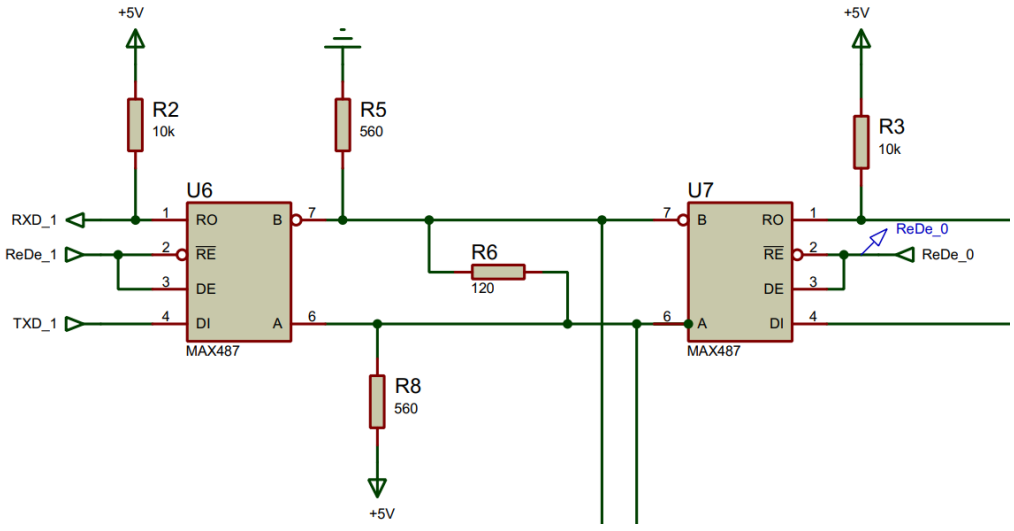
Para fins de simulação, foi possível a criação de uma porta COM virtual, a qual foi configurada devidamente no Proteus 8 e no servidor em si. A partir disso foi estabelecido um

link entre as duas partes, de modo que toda informação enviada a partir do servidor fosse transmitida pelo terminal Tx (pino 2) da porta COM, da mesma forma que as informações recebidas eletricamente, pelo terminal Rx (pino 3), fossem recebidas e lidas pelo servidor. O servidor possui ferramentas e interfaces que permitem o suporte à comunicação serial.

O circuito integrado MAX-487 apresentado à esquerda da Figura 24 realiza a conversão do sinal TTL para RS-485. No entanto, o sinal oriundo da porta COM é RS-232, portanto, visando converter RS-232 em TTL, para que esse seja convertido em RS-485, foi utilizado o componente integrado MAX-232 como intermediário.

Na Figura 25 pode ser vista a linha RS-485 de fato utilizada para a comunicação entre o servidor e os microcontroladores, a qual está sendo intermediada por dois componentes integrados MAX-487. É possível observar as duas linhas diferenciais (A e B) pelas quais os dados são transmitidos, sendo que, de acordo com os dados já abordados sobre o protocolo, o comprimento da linha pode se estender a 1220 m, permitindo comunicação a longas distâncias entre o servidor e os microcontroladores (muito adequado para o projeto, dado que os ambientes cadastrados da residência podem estar localizados consideravelmente distantes do servidor). Outro ponto importante é a existência de resistências entre as linhas diferenciais, garantindo uma maior integridade do sinal transmitido. Além disso, como característica do protocolo, podem ser conectados até 255 dispositivos ao barramento de dados, o que para o presente projeto significa expansão para até 255 ambientes para controle/monitoramento de temperatura (considerando um dispositivo RS-485 por ambiente).

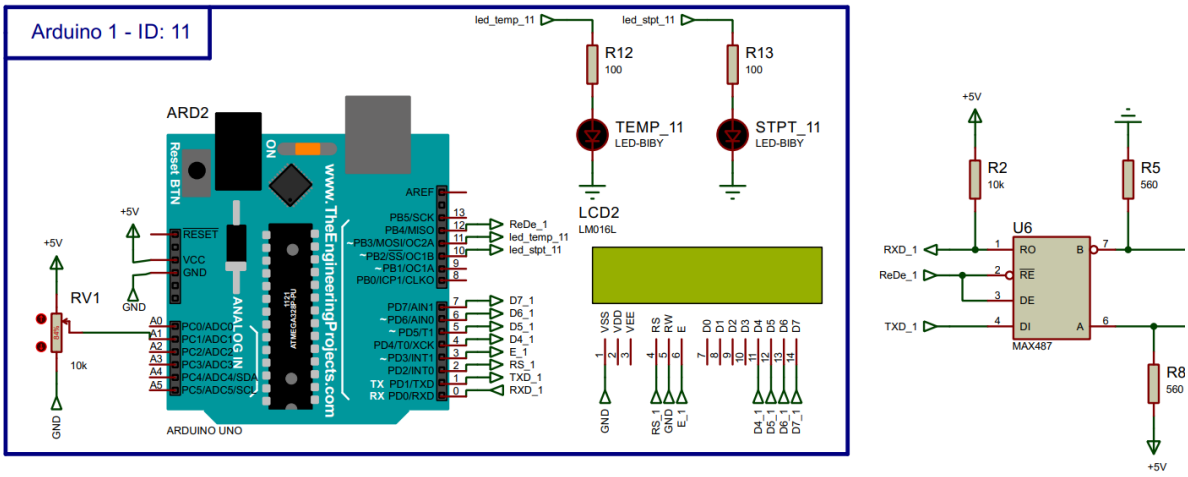
Figura 25 - Linha RS-485 para comunicação entre servidor e dispositivos de campo



Fonte: Autoria Própria

A Figura 26 apresenta o trecho final do circuito, o qual mostra a interface entre a linha RS-485 e o dispositivo de campo (arduino).

Figura 26 - Trecho do circuito referente ao microcontrolador (arduino)



Fonte: Autoria Própria

O arduino não se comunica diretamente com RS-485, tornando necessária a presença do MAX-487 para realizar a conversão de RS-485 para TTL, o qual pode ser lido pelo arduino. Para a devida conexão, os pinos RO (*Receiver Output*) e DI (*Driver Input*) do MAX-487 se conectam às portas Rx e Tx do arduino, respectivamente. Com isso, o circuito explicado até esse ponto permite a devida comunicação *half-duplex* entre o servidor e os dispositivos de campo.

A implementação mais desenvolvida do projeto conteria elementos (sensores e dispositivos de controle de temperatura de um ambiente) intermediados pelo arduino, o qual realiza o papel de dispositivo de comando e processamento. No entanto, para fins de exemplificação, foi utilizado um potenciômetro conectado a uma das entradas analógicas do arduino visando simular um valor de temperatura lido por um dispositivo, o qual é um dado repassado para o servidor. Além disso, simbolicamente foram utilizados dois *leds* (TEMP_ e STPT_) para indicar o tipo de comunicação/operação ocorrendo em determinado momento entre o servidor e o arduino (referente à dados de temperatura ou *setpoint* desejado para um ambiente). O *display* LCD, por fim, permitiu o monitoramento de informações referentes ao processamento interno da requisição recebida pelo arduino, indicando o momento que o microcontrolador recebeu uma requisição, além de evidenciar o tipo da requisição e valores associados a mesma.

Em relação à programação dos microcontroladores, o código implementado é definido no Apêndice M.

4.2 MODELAGEM DA BASE DE DADOS

Como já introduzido, a base de dados é o local em que são armazenadas todas as informações relevantes acerca dos ambientes cadastrados no projeto, para controle e monitoramento de temperatura. Além disso, também são armazenados dados referentes aos dispositivos de refrigeração/aquecimento comandados pelo microcontrolador presente em cada ambiente.

Os parâmetros utilizados para a caracterização de um ambiente foram definidos em uma tabela nomeada *ambiente* na base de dados, a qual possui os campos especificados no Quadro 8.

Quadro 8 - Informações a respeito dos ambientes na base de dados

Campo	Tipo de dados	Descrição
ambient_id	INTEGER	Fornece uma identificação única a cada ambiente
name	VARCHAR(50)	Nome atribuído a cada ambiente
temperature	INTEGER	Valor de temperatura atual monitorada em cada ambiente
setpoint	INTEGER	Valor de temperatura que o usuário deseja em um determinado ambiente

Fonte: Autoria Própria

Já para a caracterização dos dispositivos de refrigeração/aquecimento, tem-se a tabela *devices*, cujos campos estão especificados no Quadro 9.

Quadro 9 - Informações a respeito dos dispositivos de refrigeração/aquecimento na base de dados

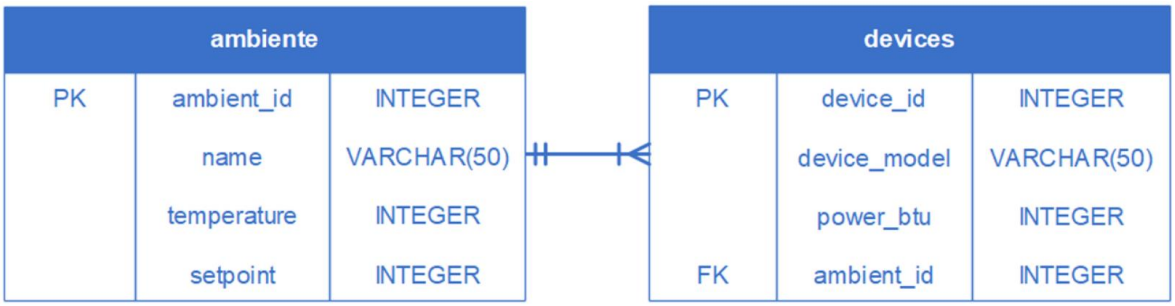
Campo	Tipo de dados	Descrição
device_id	INTEGER	Fornece uma identificação única a cada dispositivo de refrigeração/aquecimento
device_model	VARCHAR(50)	Modelo de cada dispositivo de refrigeração/aquecimento
power_btu	INTEGER	Potência de cada dispositivo de refrigeração/aquecimento em BTU
ambient_id	INTEGER	ID do ambiente ao qual o dispositivo de aquecimento/refrigeração faz parte

Fonte: Autoria Própria

A partir da definição das informações contidas na base de dados, pode ser observado na Figura 27 o diagrama de entidade e relacionamento entre as tabelas *ambiente* e *devices*. No diagrama é possível visualizar os campos apresentados, as informações em relação ao tipo de *key* que cada campo assume e a relação existente entre as duas tabelas.

Em relação às *keys* apresentadas, tem-se que *ambient_id* e *device_id* são chaves primárias, ou seja, são chaves únicas responsáveis por identificar cada um dos elementos/registros de suas respectivas tabelas. Além disso, tem-se *ambient_id* na tabela *devices*, que é uma *foreign key*, ou seja, é uma chave que faz um *link* com a tabela *ambiente*, visando identificar a qual ambiente o dispositivo pertence. Nesse contexto, pode ser feita a relação de um para muitos entre as tabelas definidas, ou seja, um ambiente pode ter mais de um dispositivo de refrigeração/aquecimento, no entanto, cada dispositivo só pode se encontrar em um ambiente. Essa relação é apresentada pelo diagrama.

Figura 27 - Diagrama de entidade e relacionamento para a base de dados



Fonte: Autoria Própria

Em relação à programação do servidor, o código que trata da manipulação e consulta da base de dados é definido no Apêndice K.

4.3 *FRAME* DAS MENSAGENS PARA A TRANSMISSÃO DE DADOS VIA RS-485

Para que os dados sejam transmitidos de forma adequada, do emissor ao receptor, de modo que esse saiba como interpretar os dados da mensagem, é necessária a implementação de um *frame*.

Do mesmo modo que protocolos de comunicação conhecidos, como o RS-232, possuem um padrão bem definido de como os bits de dados são organizados, foi criada uma estrutura para o presente projeto contendo 7 bytes de dados. Cada byte representa uma parte da informação da mensagem, sendo transmitida de acordo com o padrão ASC-II.

A Figura 28 mostra a representação geral do frame desenvolvido. A função dos bytes é descritas a seguir:

- Bytes 0 e 1: indicam o *id* do ambiente ao qual a operação se refere;
- Byte 2: referente ao tipo de operação realizada (1 para função de solicitação de temperatura e 2 para atualização de *setpoint*);
- Bytes 3 e 4: função variável;
- Bytes 5 e 6: armazenam o valor do CRC (*Cyclic Redundancy Check*).

Figura 28 – *Frame* geral da mensagem utilizada na comunicação entre servidor e dispositivos de campo

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
PRIMEIRO CARACTERE DO ID	SEGUNDO CARACTERE DO ID	FUNÇÃO: GET TEMPERATURA OU PATCH SETPOINT	-	-	PRIMEIRO BYTE DO CRC	SEGUNDO BYTE DO CRC

Fonte: Autoria Própria

O CRC é um método de verificação de erros na transmissão de uma mensagem. A partir de uma lógica estabelecida são realizadas operações matemáticas envolvendo a mensagem a ser transmitida (no caso do projeto são 5 bytes de dados). O resultado dessas operações gera 2 bytes com um valor específico, os quais são anexados no final da mensagem de 5 bytes e enviados ao destino. Quando a mensagem de 7 bytes chega ao destino, o mesmo realiza o cálculo do CRC tendo como base os 5 primeiros bytes de dados. Feito isso, o valor obtido é comparado com os 2 bytes anexados no final da mensagem, de modo que caso ambos sejam equivalentes, é visto que a mensagem foi transmitida sem erros.

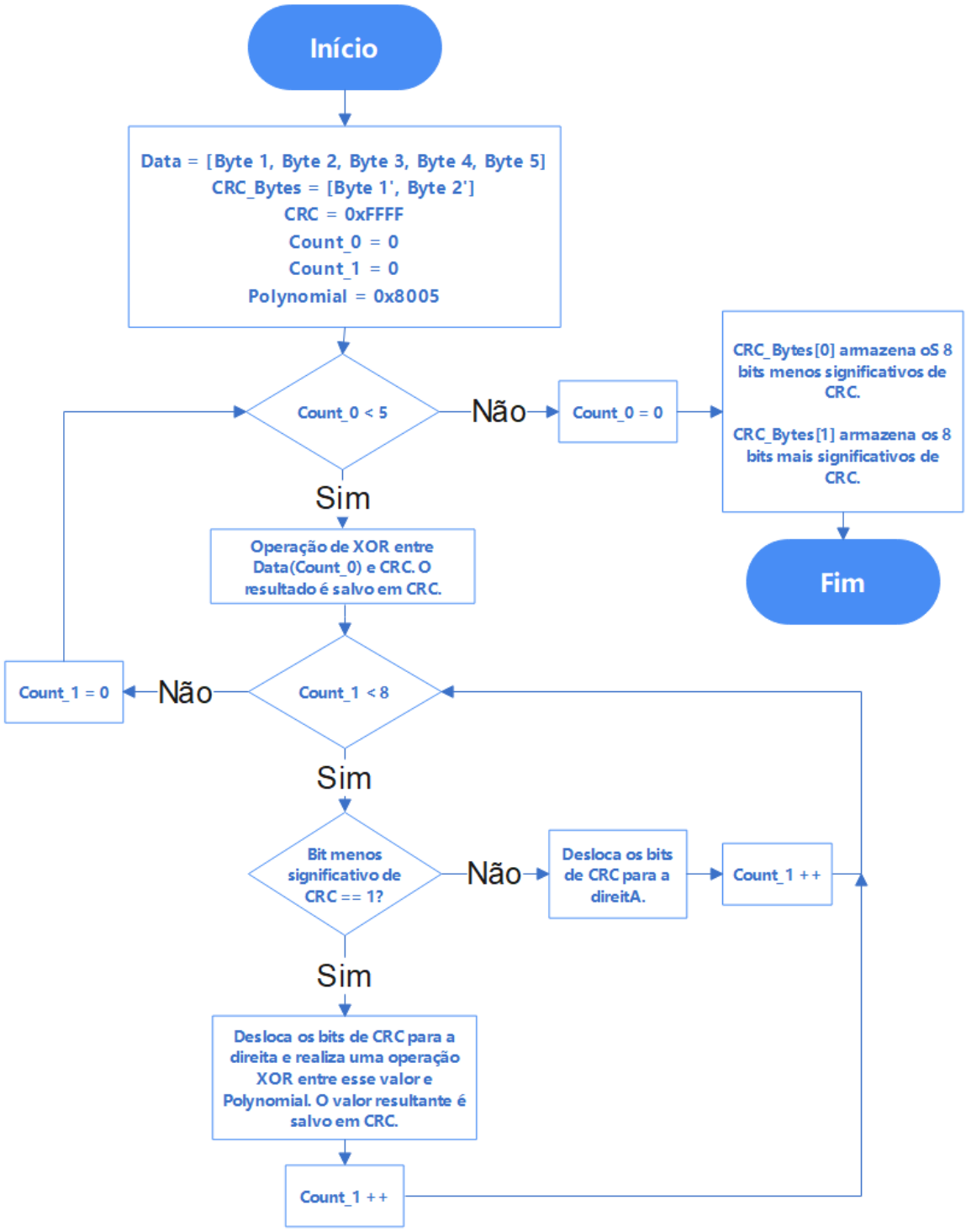
Em relação à programação do servidor, o código que apresenta as funções relacionadas ao CRC e ao envio e recebimento de dados é definido no Apêndice J.

4.3.1 Cálculo do CRC

Como apresentado por Nery (2020), o CRC é calculado com base nos bits de dados a serem transmitidos e um polinômio gerador. O polinômio é uma função G(x), cujos coeficientes são representados por números binários, formando uma sequência binária. Esse polinômio determina o tamanho do CRC gerado, sendo que quanto maior seu valor, maior sua capacidade de detecção de erros.

O cálculo de CRC implementado no projeto é baseado no CRC definido pelo protocolo Modbus RTU. Baseado em WEG (2014), pode ser explicada a lógica de cálculo (criada via *software*), apresentada na Figura 29.

Figura 29 - Diagrama de processo para cálculo de CRC



Fonte: Autoria Própria

O cálculo do CRC utiliza 3 elementos principais, os bytes de dados (5 bytes), 2 bytes referentes ao polinômio gerador (igual a 0x8005) e 16 bits referentes a uma variável CRC, que armazena de fato os valores de CRC calculados.

Primeiramente é carregado um valor inicial à variável CRC, igual a 0xFFFF. Em seguida é realizada uma operação de XOR entre os 8 bits menos significativos de CRC e o primeiro byte de dados da mensagem sobre a qual será calculado o CRC. O resultado da operação XOR é salvo na variável CRC.

O próximo passo é a verificação do bit menos significativo de CRC. Caso o bit menos significativo seja igual a 1, todos os bits de CRC são deslocados para a direita (bit menos significativo é retirado de CRC). No entanto, se o bit menos significativo for igual a 0, também ocorre o deslocamento para a direita dos bits de CRC, sendo realizada uma operação de XOR entre esse valor já deslocado e o polinômio gerador previamente definido. Feito esse processo, novamente é avaliado o bit menos significativo de CRC, sendo repetida a operação descrita. Como a lógica de cálculo visa contemplar todos os 8 bits do primeiro byte de dados analisado, a operação é realizada 8 vezes.

Contemplados os 8 bits do primeiro byte de dados, é passado para o próximo byte, sendo novamente realizado uma operação de XOR entre o polinômio gerador e o CRC, o qual nesse ponto está com o valor totalmente modificado quando comparado ao valor que possuía no momento inicial, em que foi feita a operação com o primeiro byte de dados. Feito isso ocorre novamente a iteração sobre os 8 bits do CRC. O processo descrito se repete até o momento em que os 5 bytes de dados forem contemplados na operação. Com isso, é gerado um valor de CRC de 16 bits, cujo cálculo considerou cada bit dos 5 bytes de dados, os quais foram manipulados em operações realizadas com um polinômio previamente definido. Esse processo garante a unicidade do CRC para os valores de dados utilizados, que é justamente o intuito desse mecanismo de verificação de erros. Caso alguns bits dos bytes de dados fossem diferentes, o resultado de CRC já não seria o mesmo.

Para o valor de CRC de 16 bits calculados, são considerados os 8 bits menos significativos e os 8 mais significativos como sendo o primeiro byte e o segundo byte de CRC, respectivamente. Esses 2 bytes são adicionados ao final dos 5 bytes de dados para que a mensagem seja transmitida adequadamente.

4.3.2 *Frame* da mensagem para a solicitação de temperatura aos dispositivos de campo

Para que a temperatura de cada ambiente seja atualizada na base de dados, o servidor envia mensagens aos dispositivos de campo realizando essa solicitação. A Figura 30 mostra

o *frame* dessa mensagem, sendo constituída por 7 bytes, cujas funções podem ser vistas na imagem.

Figura 30 - *Frame* geral da mensagem enviada ao dispositivo de campo solicitando temperatura

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
PRIMEIRO CARACTERE DO ID	SEGUNDO CARACTERE DO ID	FUNÇÃO CARACTERE 1 PARA REQUISIÇÃO GET TEMPERATURA	BYTE PREENCHIDO COM 0	BYTE PREENCHIDO COM 0	PRIMEIRO BYTE DO CRC	SEGUNDO BYTE DO CRC

Fonte: Autoria Própria

As funções dos bytes 0, 1, 2, 5 e 6 já foram introduzidas. O byte 2, como se trata de uma função de solicitação de temperatura, é preenchido com o caractere 1. Os bytes 3 e 4, por sua vez, são preenchidos com o caractere 0, sendo os bytes posteriormente carregados com o valor de temperatura atual lida pelo dispositivo de campo.

O *frame* da mensagem de resposta do dispositivo de campo ao servidor, com o valor de temperatura lido, pode ser visto na Figura 31, de modo que os bytes 3 e 4 contém o valor de temperatura atual do dispositivo (decimal e unidade, respectivamente).

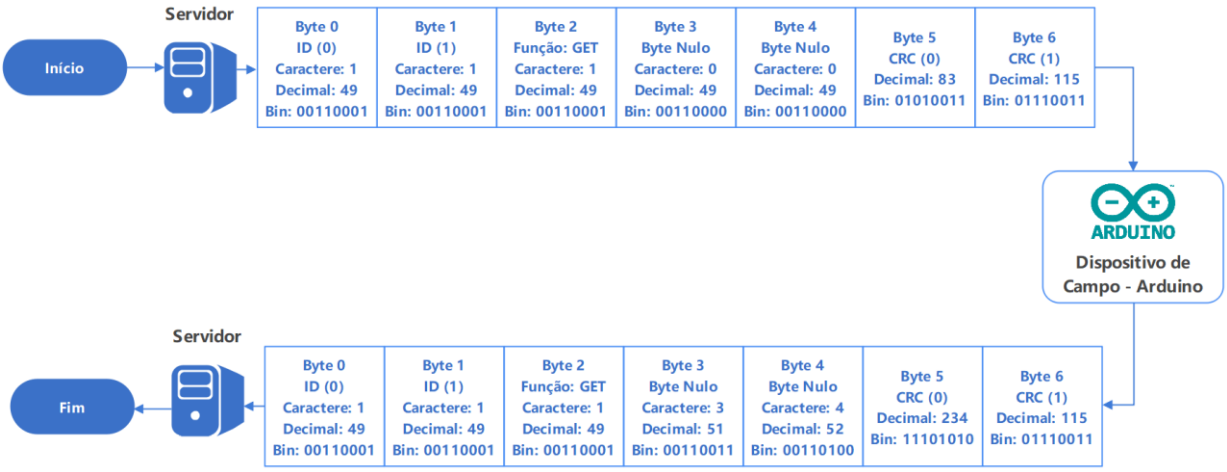
Figura 31 – *Frame* da mensagem enviada ao servidor com valor de temperatura atual medida pelo dispositivo de campo

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
PRIMEIRO CARACTERE DO ID	SEGUNDO CARACTERE DO ID	FUNÇÃO: CARACTERE 1 PARA REQUISIÇÃO GET TEMPERATURA	PRIMEIRO CARACTERE DO VALOR DE TEMPERATURA	SEGUNDO CARACTERE DO VALOR DE TEMPERATURA	PRIMEIRO BYTE DO CRC	SEGUNDO BYTE DO CRC

Fonte: Autoria Própria

A Figura 32 mostra um exemplo com valores reais de como seria estabelecida essa comunicação entre o servidor e um dispositivo de campo. Para o exemplo dado o servidor solicitou a temperatura do ambiente com *id* 11, o qual, por sua vez, retornou a temperatura lida (34ºC), sendo armazenado nos bytes 3 e 4 os caracteres 3 e 4, respectivamente.

Figura 32 - Exemplo da estrutura dos dados da comunicação para solicitação de temperatura a um dispositivo de campo



Fonte: Autoria Própria

4.3.3 *Frame* da mensagem para a atualização de *setpoint* dos dispositivos de campo

Quando o usuário realiza uma solicitação de atualização de *setpoint*, esse valor é enviado do servidor ao dispositivo de campo, podendo ser vista na Figura 33 o *frame* completo da mensagem.

Figura 33 - *Frame* da mensagem enviada ao dispositivo de campo para atualização de *setpoint*

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
PRIMEIRO CARACTERE DO ID	SEGUNDO CARACTERE DO ID	FUNÇÃO: CARACTERE 2 PARA REQUISIÇÃO PATCH SETPOINT	PRIMEIRO CARACTERE COM SETPOINT DESEJADO	SEGUNDO CARACTERE COM SETPOINT DESEJADO	PRIMEIRO BYTE DO CRC	SEGUNDO BYTE DO CRC

Fonte: Autoria Própria

Mais uma vez a função dos bytes 0, 1, 2, 5 e 6 já foram apresentadas. Nesse caso, por se tratar de uma operação de atualização de *setpoint* de temperatura, o byte 2 é preenchido com o caractere 2. Os bytes 3 e 4, por sua vez, armazenam o valor do novo *setpoint* desejado para o dispositivo de campo.

Recebida a solicitação de atualização de *setpoint* de temperatura e realizado o processo de armazenamento do novo *setpoint*, o dispositivo de campo responde ao servidor de acordo

com a Figura 34. Os bytes 3 e 4 devem ser preenchidos com o caractere 1 para indicar o sucesso da operação.

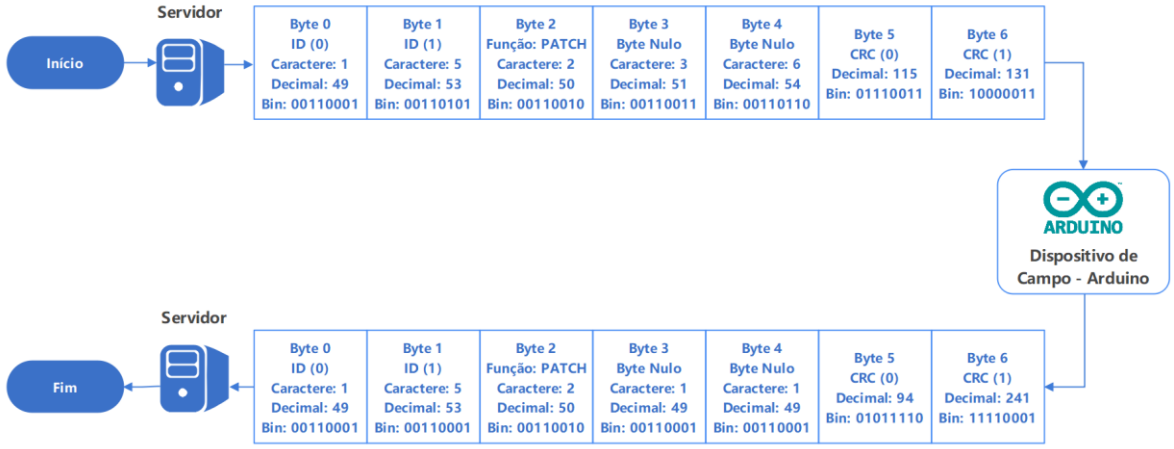
Figura 34 – *Frame* da mensagem enviada ao servidor para confirmar atualização de temperatura

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
PRIMEIRO CARACTERE DO ID	SEGUNDO CARACTERE DO ID	FUNÇÃO: CARACTERE 2 PARA REQUISIÇÃO PATCH SETPOINT	CARACTERE INDICANDO SUCESSO DA OPERAÇÃO: SE FOR IGUAL A 1 -> SUCESSO	CARACTERE INDICANDO SUCESSO DA OPERAÇÃO: SE FOR IGUAL A 1 -> SUCESSO	PRIMEIRO BYTE DO CRC	SEGUNDO BYTE DO CRC

Fonte: Autoria Própria

A Figura 35 mostra um exemplo com valores reais da estrutura de comunicação entre o servidor e um dispositivo de campo, em que o primeiro solicita ao segundo a atualização do *setpoint* de temperatura para 36ºC, como pode ser observado pelos bytes 3 e 4 da mensagem enviada pelo servidor.

Figura 35 - Exemplo da estrutura dos dados da comunicação para atualização de *setpoint* de um dispositivo de campo



Fonte: Autoria Própria

4.4 REQUISIÇÕES E OPERAÇÕES DESENVOLVIDAS

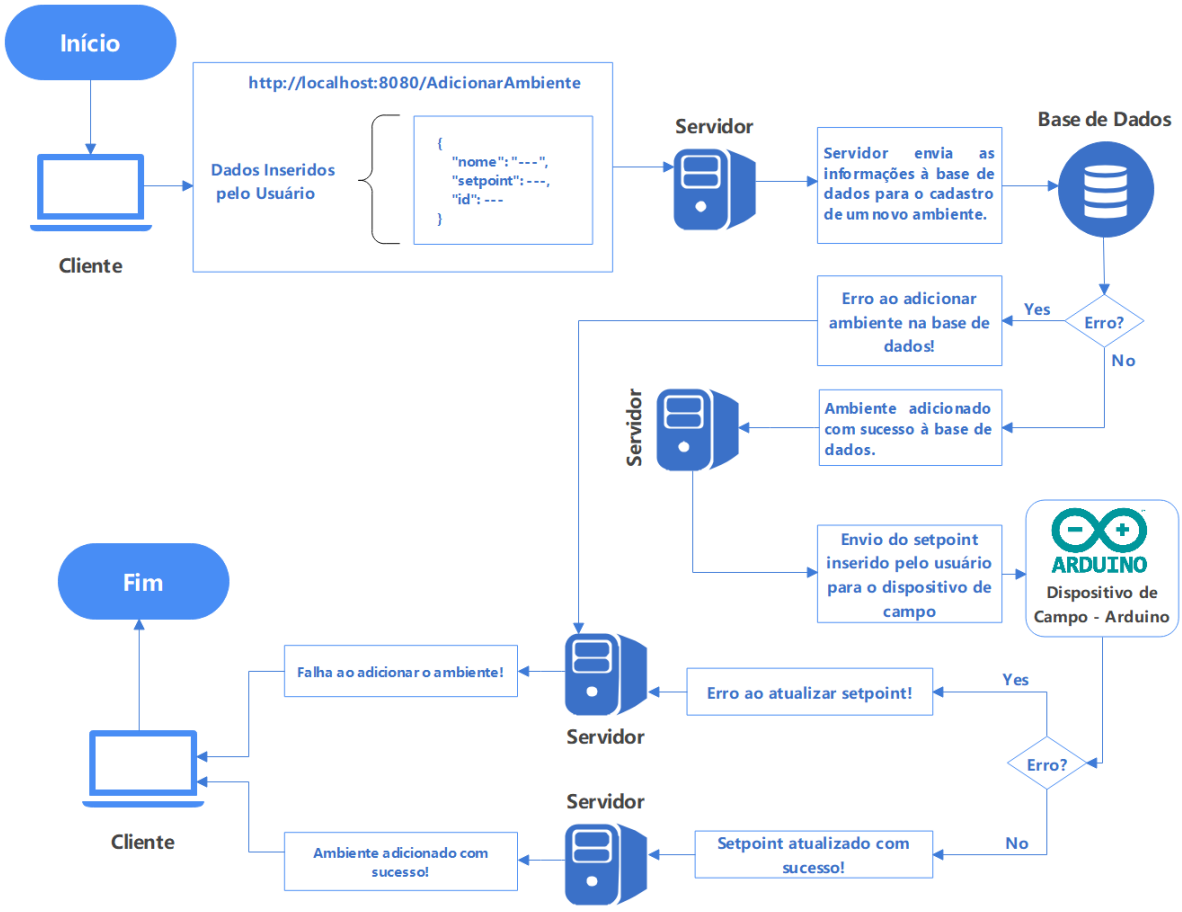
As requisições HTTP e os processos internos realizados possuem cada um uma lógica distinta, envolvendo a comunicação entre todas as partes do sistema (cliente, servidor, base

de dados e dispositivos de campo), havendo em cada caso um tratamento de informações específico, garantindo que todo o sistema atenda o usuário de forma adequada. É essencial para o projeto o entendimento de como cada operação ocorre. Portanto, serão introduzidos diagramas explicativos. Além disso, serão apresentadas as possíveis respostas ao usuário, para cada requisição, com o respectivo *status*.

4.4.1 Requisição para adicionar novo ambiente ao sistema

A Figura 36 mostra o diagrama representativo do processo de cadastro de um novo ambiente ao sistema de controle e monitoramento de temperatura.

Figura 36 - Diagrama do processo de adição de novo ambiente ao sistema



Fonte: Autoria Própria

A operação começa por meio do usuário, ao acessar a rota dada por:

```
POST: http://localhost:8080/AdicionarAmbiente
```

Os dados necessários são inseridos no corpo da requisição para que esse novo ambiente seja cadastrado no sistema, sendo eles os parâmetros *nome* (nome dado ao ambiente), *setpoint* (setpoint de temperatura desejado) e *id* (valor numérico único responsável por identificar o ambiente).

O servidor recebe os dados, realizando uma análise para verificação de erros antes de prosseguir, ou seja, é avaliado se os parâmetros inseridos estão todos de acordo com o esperado. Caso algum parâmetro possua um tipo inesperado (alguma letra inserida no campo de *setpoint*, por exemplo), ou esteja vazio, é retornado um erro ao usuário, inviabilizando a conclusão da operação. Foi definida também uma faixa de valores aceitáveis para o *setpoint* de temperatura, de 10ºC a 40ºC, sendo estabelecido automaticamente o valor padrão de 20ºC caso o valor inserido pelo usuário esteja fora do intervalo.

Sendo verificados os parâmetros inseridos, o servidor recebe os dados propriamente e se comunica com a base de dados, armazenando o novo ambiente e suas informações na mesma. Caso ocorra algum erro na operação, o servidor recebe a falha e retorna ao usuário que ocorreu algum erro ao cadastrar o ambiente na base de dados.

Cadastrado com sucesso o novo ambiente na base de dados, o servidor realiza o envio do *setpoint* inserido pelo usuário ao dispositivo de campo responsável pelo controle de temperatura, sendo reportado ao usuário caso ocorra algum erro. Em relação ao recebimento da resposta do dispositivo de campo, qualquer erro que ocorra pode ser observado a partir da mensagem de 7 bytes recebida pelo servidor, podendo ocorrer, principalmente, a não correspondência do valor de CRC ou valores dos bytes 3 e 4 diferentes de 1 (indicando que a operação não ocorreu com sucesso). Com isso será retornado ao usuário uma mensagem de falha na operação.

Caso todo o processo ocorra sem falhas, o servidor retorna ao cliente que o cadastro do novo ambiente ocorreu com sucesso.

As possíveis respostas ao usuário para essa requisição podem ser observadas no Quadro 10.

Quadro 10 - Possíveis respostas para a requisição de adição de novo ambiente ao sistema

Status	Conteúdo da resposta
400	*To insert a new ambient, it must be a valid name!*
400	*To insert a new ambient, it must be a valid setpoint!*
500	*Error to register new ambient! Problem updating database!*
500	*Error to update setpoint! Confirmation bytes content (bytes 3 and 4) received from field device are not the expected!*
500	*Error to update Setpoint! Confirmation message from field device is corrupted (CRC mismatch)!*
500	*Error to update setpoint! Problem to send data to field device*.
500	*Error to update setpoint! Problem receiving data from field device.*
201	*Temperature of (temperature value) is out of appropriate range, default setpoint value of 20ºC was set. Ambient registered succesfully!*
201	*New ambient registered succesfully!*

Fonte: Autoria Própria

Os códigos de *status* HTTP podem assumir diferentes valores: 400 (solicitação feita pelo cliente é inválida), 201 (solicitação bem-sucedida com criação de novo recurso), 200 (solicitação bem-sucedida) e 500 (erro interno no servidor). Esses valores são uma forma de expressar a natureza do que ocorreu com a requisição, no entanto, os detalhes são fornecidos por meio da mensagem contida no corpo da resposta.

Considerando as duas primeiras linhas do Quadro 10, quando é inserido um nome ou um *setpoint* inválido, o conteúdo da mensagem de resposta ao cliente é diferente em cada situação, uma vez que deve ser informado ao cliente especificamente qual dos parâmetros gerou a falha. No entanto, a falha tem a mesma natureza para ambos os casos, já que são erros gerados devido ao cliente, o que justifica a utilização do mesmo código *status* (400).

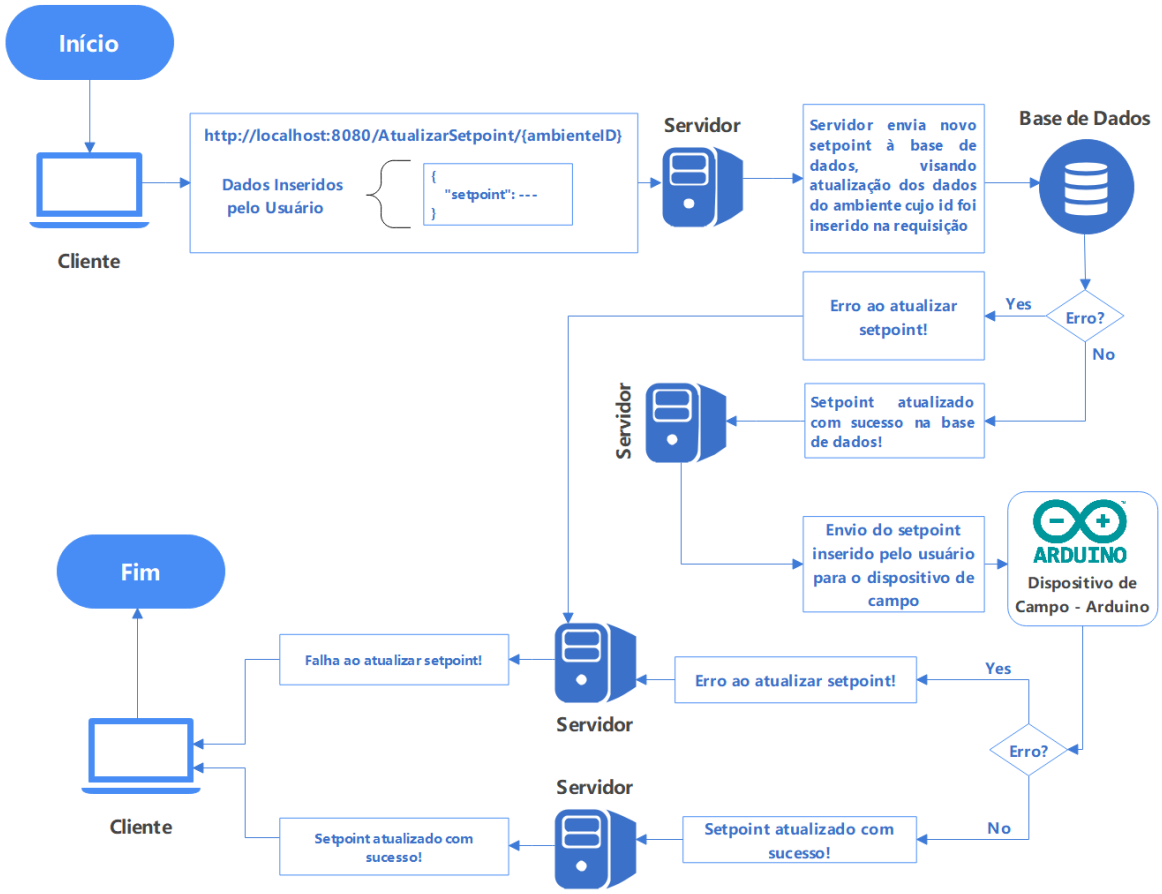
Essa explicação em relação a utilização de valores iguais de códigos *status* se mantém para os Quadros 11, 12, 13, 14 e 15.

Em relação à programação do servidor, o código que apresenta o *handler* que trata a requisição é definido no Apêndice C.

4.4.2 Requisição para atualização de *setpoint* de um ambiente

A Figura 37 mostra o diagrama representativo do processo de atualização de *setpoint* de um ambiente já cadastrado no sistema.

Figura 37 - Diagrama do processo para atualização de *setpoint* de um ambiente



Fonte: Autoria Própria

O processo apresentado é muito semelhante ao de cadastro de um novo ambiente, tendo em vista que ao adicionar um novo ambiente, é adicionado com o mesmo um valor de *setpoint*.

Para que o usuário realize a operação para alterar o *setpoint* de um ambiente, o mesmo deve utilizar a seguinte rota:

```
PATCH: http://localhost:8080/AtualizarSetpoint/{ambienteID}
```

A rota apresentada deve conter no final o valor do *id* do ambiente ao qual é desejada a atualização do *setpoint*. Além disso, no corpo da requisição deve ser informado o valor do *setpoint* desejado em si.

Recebidos os dados, o servidor avalia se o *setpoint* inserido é um valor válido e se o *id* inserido na rota corresponde ao de algum ambiente cadastrado, sendo retornado algum erro ao usuário em caso de divergência. Novamente, o *setpoint* inserido deve estar na faixa especificada, sendo configurado como 20ºC caso não esteja.

Como mostrado para a requisição de cadastro de um novo ambiente, o servidor se comunica com a base de dados, enviando os dados à mesma. Caso ocorra algum erro, o mesmo é informado ao servidor, que por sua vez é retornado ao usuário, indicando falha na atualização.

Em seguida, o servidor envia o novo *setpoint* ao ambiente com *id* especificado na requisição, atualizando-o. Novamente, caso algum erro ocorra tanto no envio do *setpoint* quanto no recebimento da confirmação da operação, o mesmo é retornado ao servidor, informando ao usuário a falha na operação.

Não ocorrendo divergências em relação ao esperado, é enviado ao usuário uma resposta de sucesso na atualização do *setpoint* do ambiente especificado.

As possíveis respostas ao usuário para essa requisição podem ser observadas no Quadro 11.

Quadro 11 - Possíveis respostas para a requisição de atualização de *setpoint* de um ambiente

Status	Conteúdo da resposta
400	*ID informed is not associated with any ambient registered!*
400	*To update setpoint, it must be a valid setpoint!*
500	*Error do update setpoint! Problem updating database.*
500	*Error do update setpoint! Problem to send data to field device*.
500	*Error do update setpoint! Problem receiving data from field device*.
500	*Error to update setpoint! Confirmation bytes content (bytes 3 and 4) received from field device are not the expected!*
500	*Error to update setpoint! Confirmation message from field device is corrupted (CRC mismatch)!*
201	*Temperature of (temperature value) is out of appropriate range, default setpoint value of 20ºC was set. Ambient setpoint succesfully updated!*
201	*Ambient setpoint succesfully updated!*

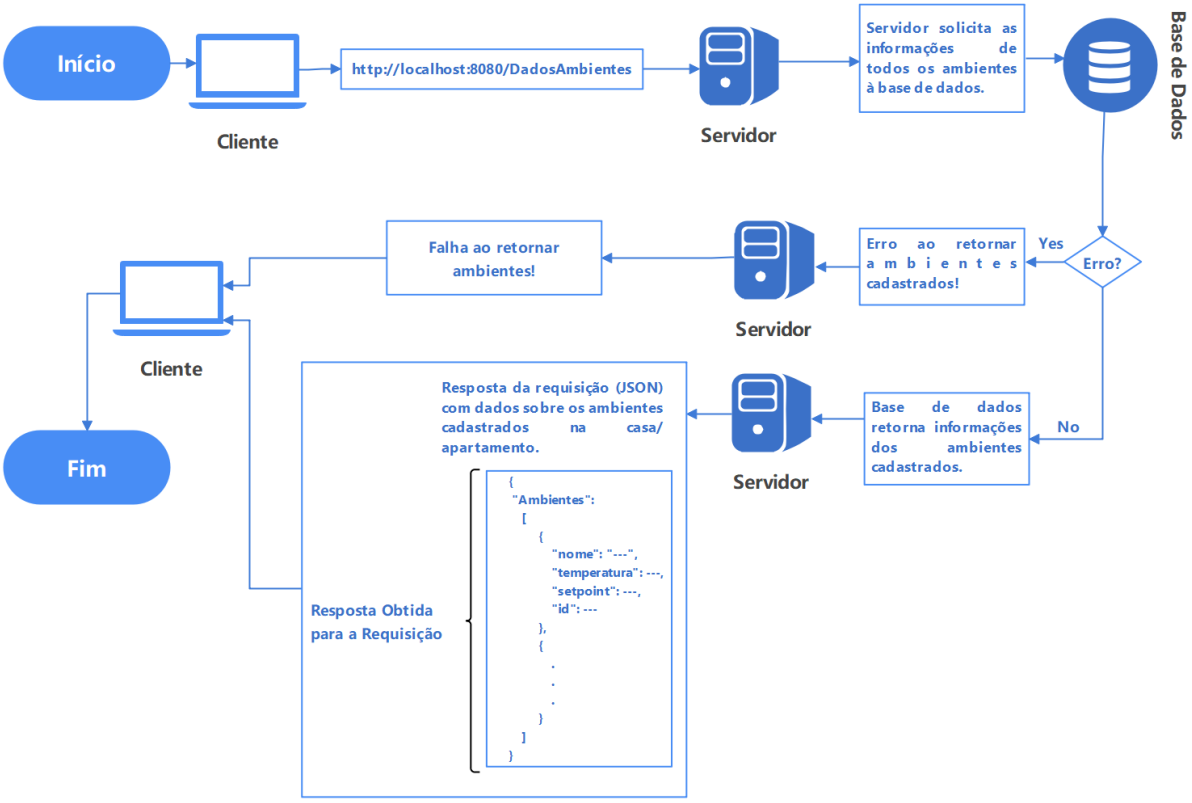
Fonte: Autoria Própria

Em relação à programação do servidor, o código que apresenta o *handler* que trata a requisição é definido no Apêndice D.

4.4.3 Requisição para obtenção de dados de todos os ambientes

A Figura 38 mostra o diagrama representativo do processo de obtenção dos dados de todos os ambientes cadastrados no sistema.

Figura 38 - Diagrama do processo para obtenção de dados gerais de todos os ambientes cadastrados



Fonte: Autoria Própria

Para que o usuário obtenha as informações de nome, temperatura, *setpoint* e *id* de todos os ambientes registrados, a seguinte rota deve ser utilizada:

```
GET: http://localhost:8080/DadosAmbientes
```

A partir da rota especificada o servidor recebe a requisição, solicitando, por sua vez, os dados de todos os ambientes cadastrados à base de dados. Caso ocorra algum erro ao resgatar os ambientes na base de dados, o mesmo é retornado ao servidor, o qual finaliza a operação enviando ao usuário uma resposta contendo o erro encontrado durante a requisição. Por outro lado, se os dados forem obtidos com sucesso, as informações de cada ambiente são enviadas ao usuário.

As possíveis respostas ao usuário para essa requisição podem ser observadas no Quadro 12.

Quadro 12 - Possíveis respostas para a requisição de dados de todos os ambientes

Status	Conteúdo da resposta
200	São retornados os dados referentes ao *ambient_id, name, temperature* e *setpoint* para todos os ambientes cadastrados na base de dados.
500	*Error to return all registered ambient data! Problem returning data from database.*

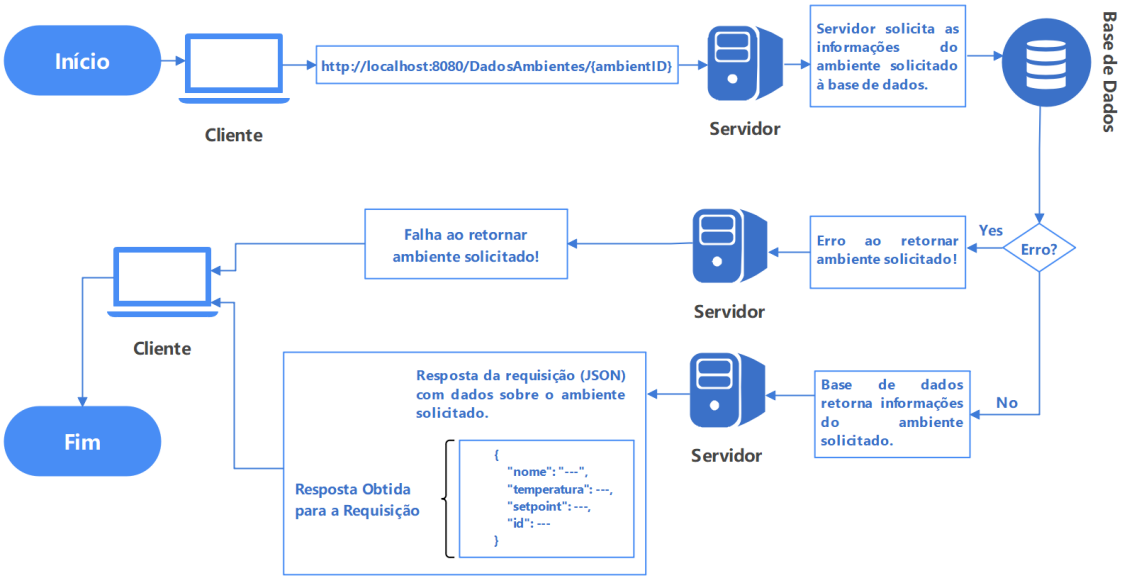
Fonte: Autoria Própria

Em relação à programação do servidor, o código que apresenta o *handler* que trata a requisição é definido no Apêndice G.

4.4.4 Requisição para obtenção de dados de um ambiente em específico

Outro tipo de requisição desenvolvida foi a que realiza a solicitação de apenas um ambiente em específico, sendo o diagrama do processo representado na Figura 39.

Figura 39 - Diagrama do processo para obtenção de dados de um ambiente específico



Fonte: Autoria Própria

Para que o usuário obtenha as informações de um ambiente em específico, a seguinte rota deve ser utilizada:

```
GET: http://localhost:8080/DadosAmbientes/{ambienteID}
```

A rota é semelhante à da requisição de solicitação de dados de todos os ambientes, sendo que a diferença é que é necessário inserir o *id* do ambiente solicitado no final da rota. Assim como a rota, o restante do processo e da verificação de erros existente nesse caso é análogo ao da solicitação dos dados de todos os ambientes. No final, caso não ocorra nenhum erro, são retornados os dados do ambiente solicitado.

As possíveis respostas ao usuário para essa requisição podem ser observadas no Quadro 13.

Quadro 13 - Possíveis respostas para a requisição de dados de um ambiente específico

Status	Conteúdo da resposta
200	São retornados os dados referentes ao *ambient_id*, *name*, *temperature* e *setpoint* para o ambiente solicitado.
400	*ID informed is not associated with any registered ambient.*
500	*Error to return registered ambient data! Problem returning data from database.*

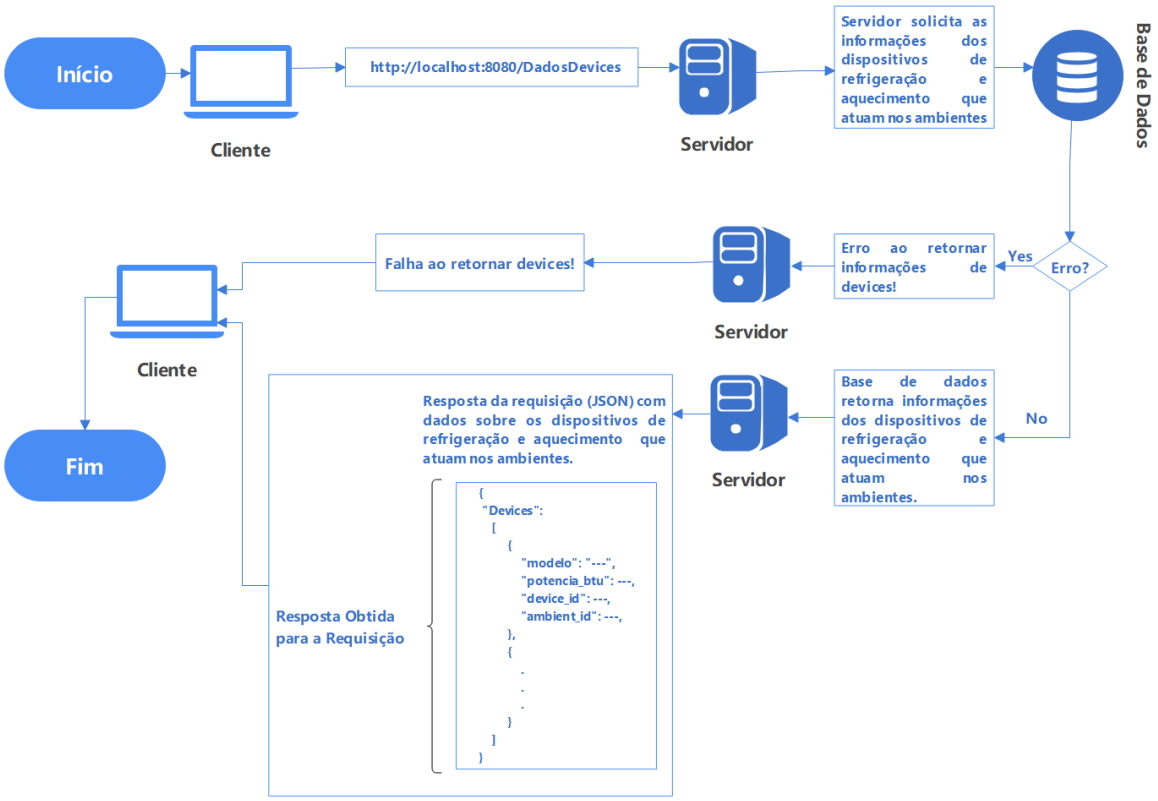
Fonte: Autoria Própria

Em relação à programação do servidor, o código que apresenta o *handler* que trata a requisição é definido no Apêndice E.

4.4.5 Requisição para obtenção de dados dos dispositivos de refrigeração/aquecimento contidos nos ambientes cadastrados

Assim como foi determinada uma rota para a solicitação de informações dos ambientes, também foi determinada uma rota que realiza a solicitação dos dados referentes aos dispositivos de refrigeração/aquecimento contidos nos ambientes cadastrados, sendo esses dados: *ambient_id*, *device_id*, *device_model* e *power_btu*. A Figura 40 apresenta o diagrama do processo.

Figura 40 - Diagrama do processo para obtenção dos dados dos dispositivos de refrigeração/aquecimento contidos nos ambientes cadastrados



Fonte: Autoria Própria

A requisição é realizada a partir da seguinte rota:

```
GET: http://localhost:8080/DadosDevices
```

O processo para a obtenção dos dados referentes a todos os dispositivos registrados para os ambientes é semelhante ao da requisição para a obtenção dos dados gerais referentes a todos os ambientes cadastrados. A diferença é que é realizada uma *query* para obtenção dos dados armazenados na tabela *devices* ao invés da tabela *ambiente*.

Caso algum erro ocorra na solicitação das informações à base de dados, o mesmo é informado ao usuário. No entanto, caso o processo seja concluído com sucesso, são retornadas as informações dos dispositivos.

As possíveis respostas ao usuário para essa requisição podem ser observadas no Quadro 14.

Quadro 14 - Possíveis respostas para a requisição de dados dos dispositivos contidos nos ambientes cadastrados

Status	Conteúdo da resposta
200	São retornados os dados referentes ao *device_id*, *ambient_id*, *device_model* e *power_btu* de todos os dispositivos cadastrados na base de dados.
500	*Error to return all devices data! Problem returning data from database.*

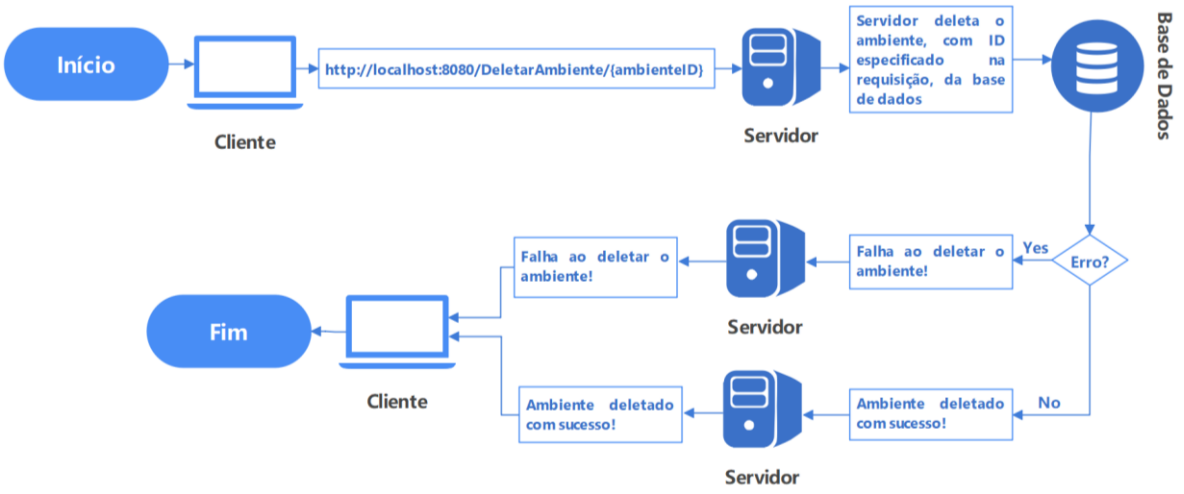
Fonte: Autoria Própria

Em relação à programação do servidor, o código que apresenta o *handler* que trata a requisição é definido no Apêndice F.

4.4.6 Requisição para remoção de um ambiente do sistema

A Figura 41 mostra o diagrama representativo do processo realizado para remoção de um ambiente registrado do sistema.

Figura 41 - Diagrama do processo para remoção de um ambiente do sistema



Fonte: Autoria Própria

A rota utilizada para a operação é dada por:

```
http://localhost:8080/DeletarAmbiente/{ambienteID}
```

Como pode ser observado, deve ser indicado no final da rota o valor numérico do *id* do ambiente cuja remoção do sistema é desejada. Caso seja inserido um *id* que não corresponde a nenhum ambiente cadastrado, é retornado um erro ao usuário.

Recebida a solicitação, o servidor se comunica com a base de dados visando a remoção do ambiente com *id* especificado (todos os *devices* associados ao ambiente em questão também são excluídos da base de dados). Caso ocorra alguma falha, é retornado o erro ao servidor, o qual indica ao usuário a falha na operação. No entanto, caso a remoção seja feita com sucesso na base de dados, é respondido ao usuário que a operação foi concluída com sucesso.

As possíveis respostas ao usuário para essa requisição podem ser observadas no Quadro 15.

Quadro 15 - Possíveis respostas para a requisição de remoção de um ambiente do sistema

Status	Conteúdo da resposta
200	*Ambient deleted succesfully!*
400	*ID informed is not associated with any registered ambient.*
500	*Error to delete ambient! Problem to delete on database.*

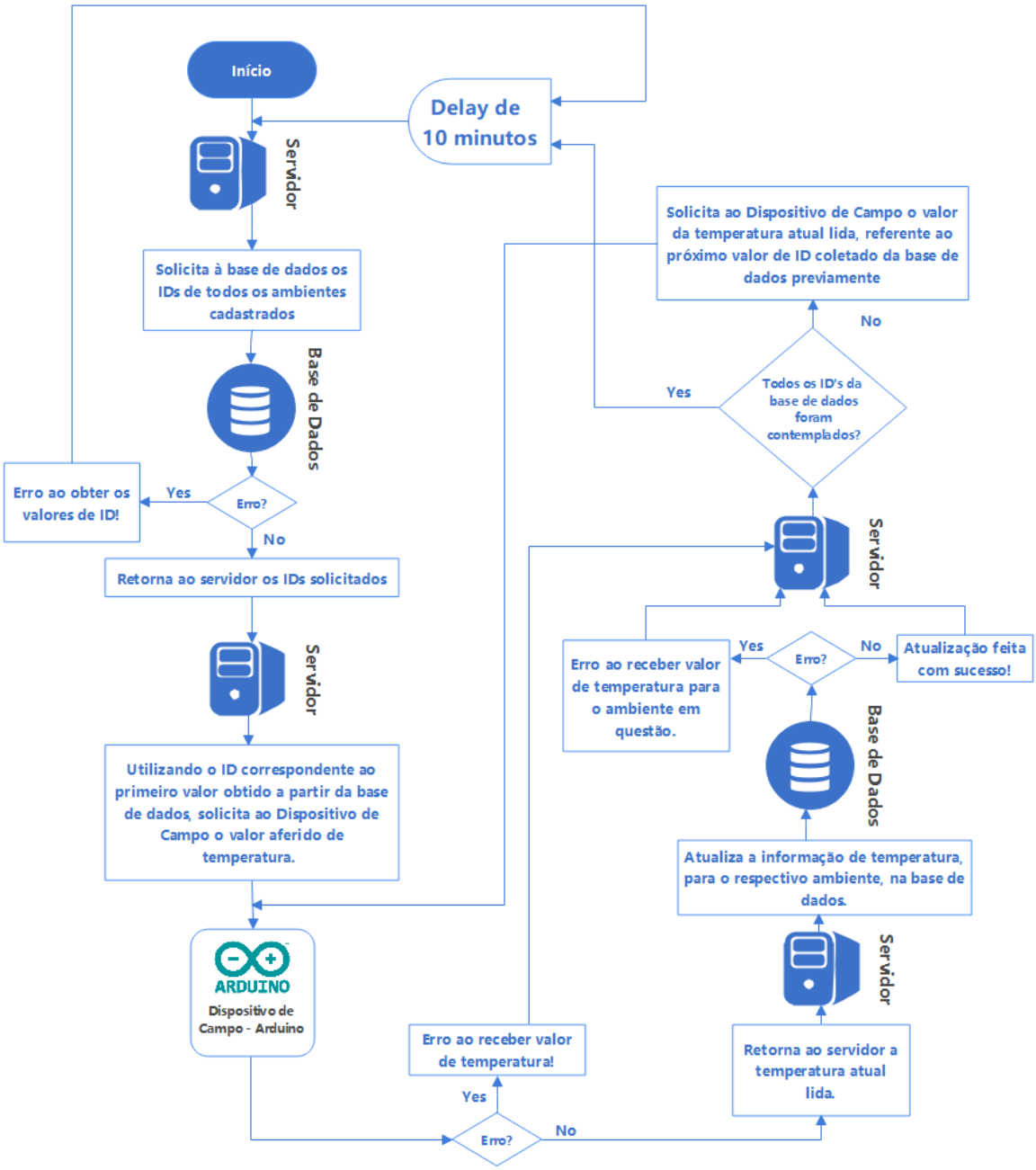
Fonte: Autoria Própria

Em relação à programação do servidor, o código que apresenta o *handler* que trata a requisição é definido no Apêndice H.

4.4.7 Lógica para atualização de temperatura

Com o intuito de manter as informações de temperatura atualizadas na base de dados, foi criada uma rotina. Essa rotina faz com que o servidor solicite, a cada período, a temperatura aferida pelos dispositivos de campo em cada um dos ambientes. A Figura 42 apresenta o diagrama com todo o processo dessa rotina em questão.

Figura 42 - Diagrama do processo para atualização de temperatura na base de dados



Fonte: Autoria Própria

O início do processo ocorre com o servidor solicitando à base de dados os valores de *id* de todos os ambientes cadastrados, armazenando todos esses valores internamente no momento que a base de dados os retorna.

Com todos os valores armazenados internamente, o servidor realiza uma solicitação de forma individualizada a cada um dos dispositivos de campo com *id* correspondente aos retornados pela base de dados inicialmente. Portanto, supondo um caso em que existam dois

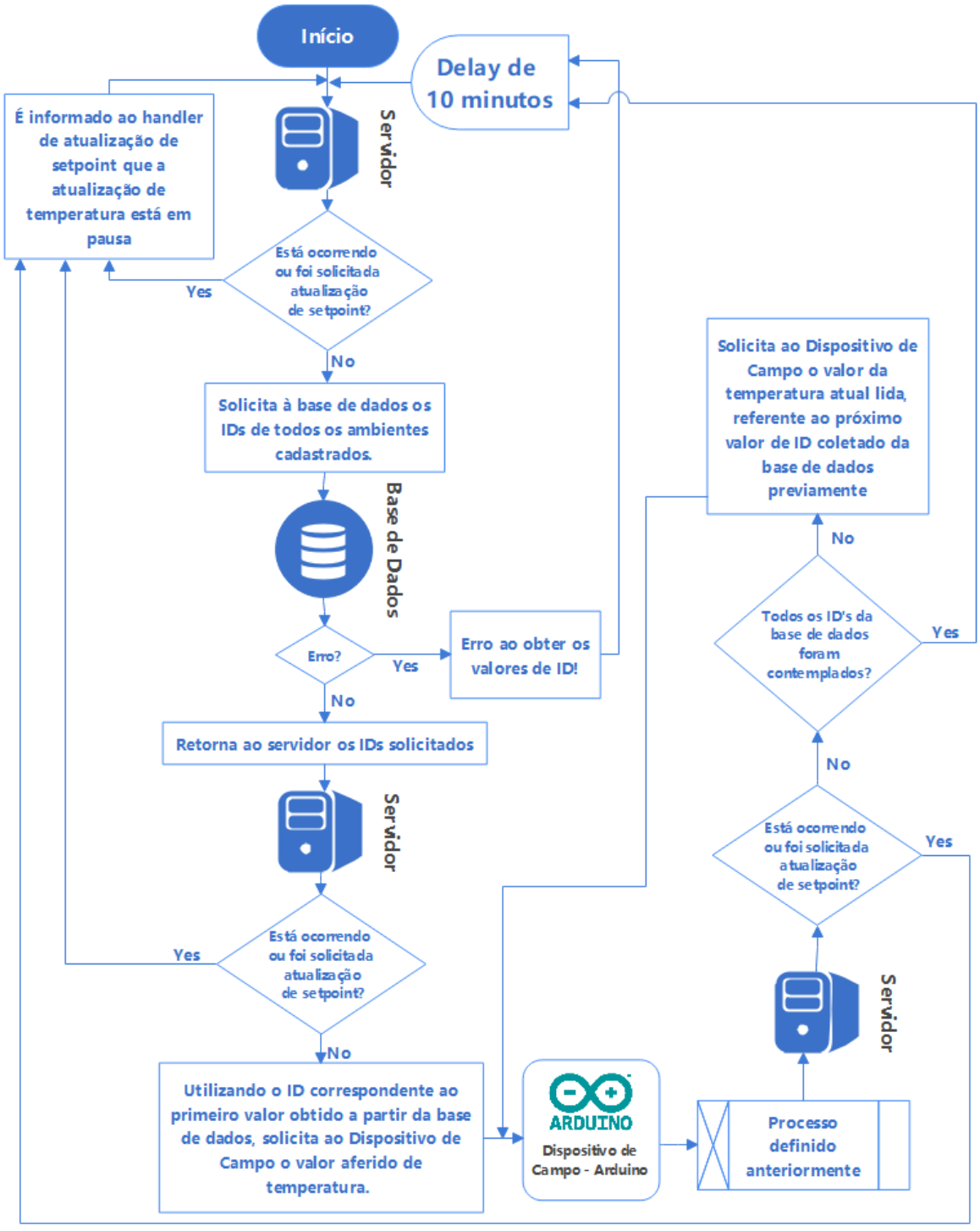
ambientes cadastrados, com *ids* 11 e 15, inicialmente o servidor solicita esses valores à base de dados e em seguida solicita ao dispositivo de campo localizado no ambiente com *id* 11 o valor de temperatura aferida. Ao retornar o valor, o servidor armazena a temperatura atualizada para o ambiente com *id* 11.

Feito esse processo uma vez, o servidor verifica se todos os *ids* já foram contemplados. Para o exemplo dado, resta ainda o ambiente com *id* 15, portanto, é feita a solicitação via RS-485 novamente, armazenando a temperatura obtida na base de dados. Sendo contemplados todos os *ids*, o servidor finaliza o processo e aguarda um período pré-estabelecido de 10 minutos antes de executar a rotina descrita novamente.

A todo momento ocorrem verificações de erro em diversas etapas do processo. A primeira verificação ocorre quando o servidor solicita à base de dados os *ids* dos ambientes cadastrados, sendo retornada uma falha caso algum problema ocorra ao longo do processo. A segunda verificação ocorre quando o servidor solicita ao dispositivo de campo o valor da temperatura aferida no momento, sendo que nesse ponto podem ocorrer erros como a divergência de valor de CRC, indicando que houve erro na transmissão da informação. O terceiro ponto é no momento que o servidor atualiza a temperatura de um ambiente em específico na base de dados, identificando algum erro nesse processo. Para cada erro foi definido um comportamento específico, sendo que para o primeiro tipo citado ocorre um reinício do processo (retornando ao *delay* de 10 minutos), para que então ocorra mais uma tentativa de atualização. Já para o segundo e para o terceiro tipo, o servidor passa para o *id* do próximo ambiente, tentando realizar a atualização de temperatura do mesmo (a não ser que todos os *ids* já tenham sido contemplados, sendo finalizada a operação e aguardando 10 minutos para começar novamente).

Também foi estabelecida uma lógica de interrupção que coloca a rotina de obtenção de temperatura dos ambientes em pausa no momento que ocorre uma requisição de atualização de *setpoint*. A Figura 43 exibe uma versão do diagrama da Figura 42, no entanto, sob uma perspectiva de interrupção caso seja solicitada a atualização do *setpoint* de determinado ambiente a partir de um usuário. A adição de um novo ambiente também acaba gerando essa interrupção, já que é inserido um determinado valor de *setpoint* no corpo da requisição.

Figura 43 - Diagrama indicativo da coordenação entre processo de atualização de temperatura e atualização de *setpoint*



Fonte: Autoria Própria

A partir da análise do fluxograma, pode ser visto que em diversos pontos da rotina de atualização de temperatura é verificado se houve uma requisição de atualização de *setpoint*. A primeira verificação ocorre logo no início da rotina, antes do servidor solicitar à base de

dados os *ids* de todos os ambientes cadastrados. A segunda ocorre antes do servidor solicitar ao dispositivo de campo a temperatura atual do ambiente. Por fim, a terceira ocorre após o dispositivo de campo retornar ao servidor a temperatura do ambiente com *id* referente à iteração do momento (verificando se houve solicitação de requisição antes de solicitar a temperatura do próximo ambiente).

Portanto, enquanto a rotina de aquisição da temperatura dos ambientes está acontecendo, a atualização de *setpoint* não ocorre, e vice-versa. Essa estrutura de interrupção criada pode ser reaproveitada em uma situação de expansão do projeto em que haja mais processos ou requisições que façam a utilização da comunicação física entre os microcontroladores e o servidor.

Em relação à programação do servidor, o código que trata da atualização de temperatura é definido no Apêndice I.

4.5 RESULTADOS E DISCUSSÃO

Um dos principais pontos para validação do funcionamento do sistema de automação desenvolvido é a avaliação a nível de tensão dos sinais que são transmitidos via RS-485.

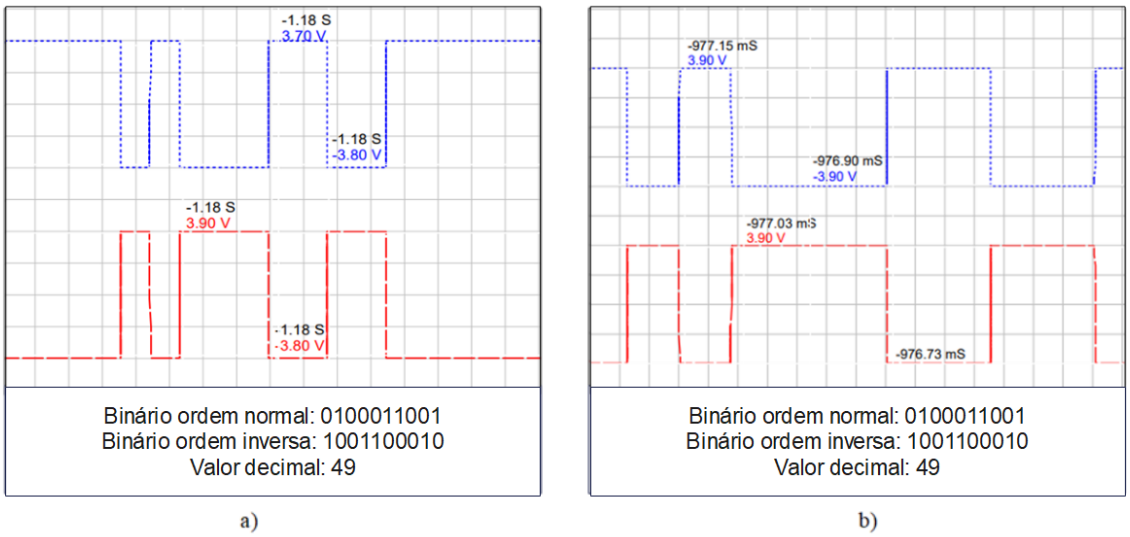
Visando validar esse funcionamento, foi medido a partir de um osciloscópio (disponível no *software* Proteus 8) as tensões nas linhas A e B para uma operação de solicitação de temperatura a um dispositivo de campo. Para a avaliação do nível lógico representado, é tido que quando a tensão $V_A - V_B$ é negativa, o nível lógico é baixo (0), ao passo que quando essa tensão é positiva, o nível lógico representado é alto (1).

O caso específico utilizado foi a solicitação ao ambiente com *id* 11 e temperatura lida igual a 34ºC, no momento avaliado. O valor de tensão lido nas linhas A e B ao longo da troca de informações entre o arduino e o servidor devem gerar os valores lógicos correspondentes ao da Figura 32, dado que os valores utilizados nesse exemplo correspondem ao caso cujas tensões foram medidas para averiguação.

As Figuras 44 a 47 apresentam as leituras de tensão do osciloscópio para o envio dos 7 bytes responsáveis por solicitar a temperatura ao dispositivo de campo, sendo que o sinal em azul é sempre referente à linha A e o sinal em vermelho é sempre referente à linha B. A Figura 44 mostra os sinais que representam os bytes 0 e 1 (responsáveis pela informação do *id* do ambiente ao qual é solicitada a temperatura). A Figura 45-a apresenta o sinal referente ao byte 2, responsável pela indicação da função solicitada (como é uma solicitação de temperatura, é uma função GET, representada pelo número 1). A Figura 45-b e a Figura 46-

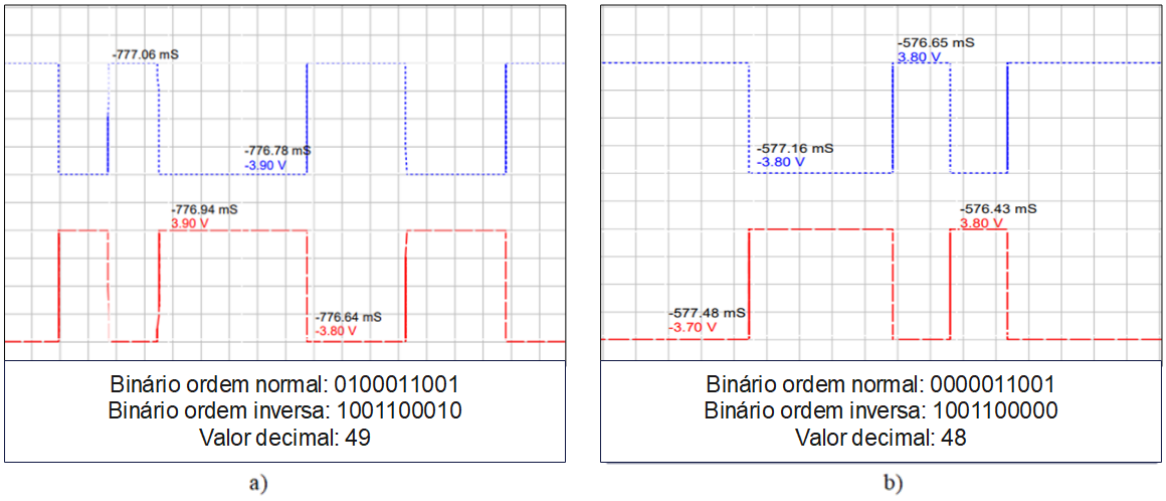
a apresentam os sinais referentes aos bytes 3 e 4, que contém valor igual a 0, como esperado. Por fim, a Figura 46-b e a Figura 47 apresentam os bytes 5 e 6, os quais representam o CRC para que seja feita a verificação de erros na transmissão da mensagem.

Figura 44 - Tensão RS-485 em uma requisição de solicitação de temperatura: a) Byte 0; b) Byte 1



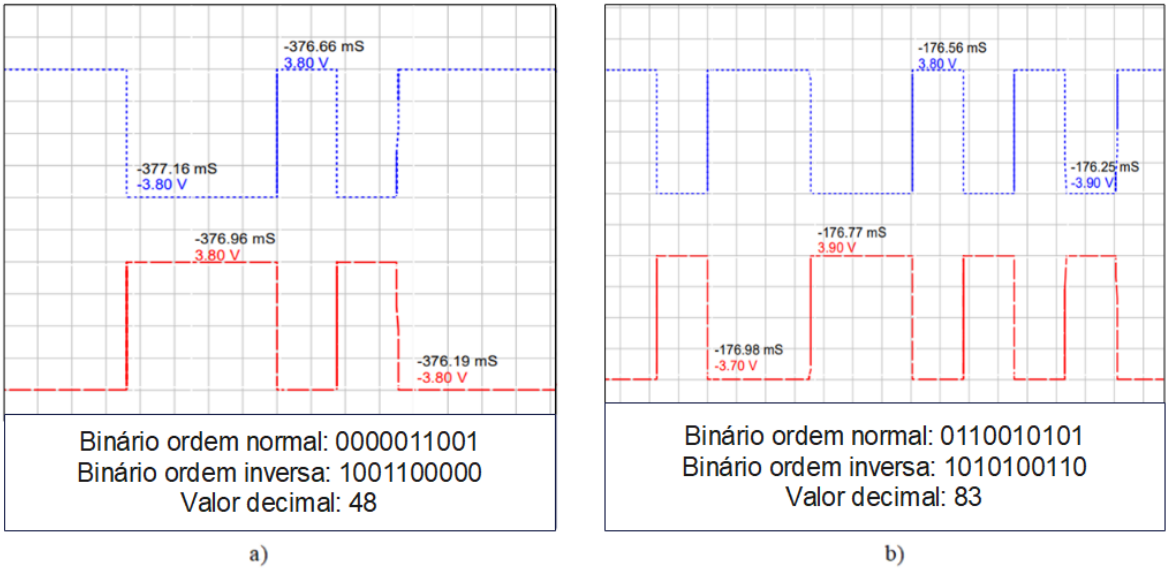
Fonte: Autoria Própria

Figura 45 - Tensão RS-485 em uma requisição de solicitação de temperatura: a) Byte 2; b) Byte 3



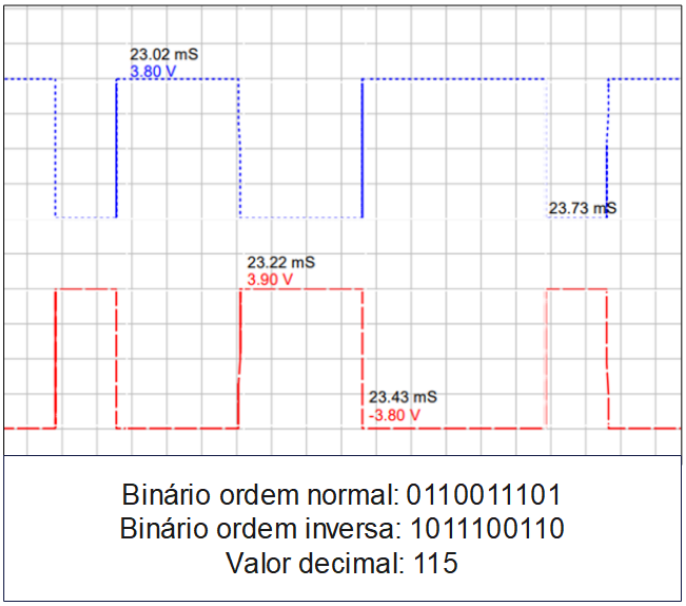
Fonte: Autoria Própria

Figura 46 - Tensão RS-485 em uma requisição de solicitação de temperatura: a) Byte 4; b) Byte 5



Fonte: Autoria Própria

Figura 47 - Tensão RS-485 em uma requisição de solicitação de temperatura: Byte 6

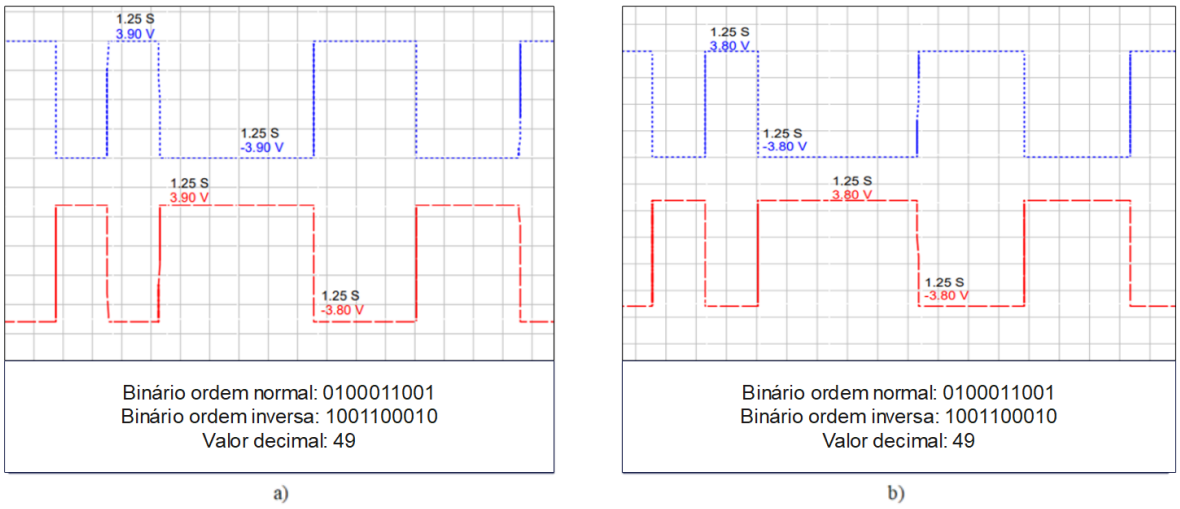


Fonte: Autoria Própria

As Figuras 48 a 51 apresentam, por sua vez, os sinais lidos pelo osciloscópio para os 7 bytes que contém a resposta do dispositivo de campo, informando ao servidor a temperatura solicitada. A Figura 48 mostra os sinais que representam os bytes 0 e 1 (responsáveis pela informação do *id* do ambiente ao qual é solicitada a temperatura). A Figura 49-a apresenta o sinal referente ao byte 2, responsável pela indicação da função solicitada (como é uma

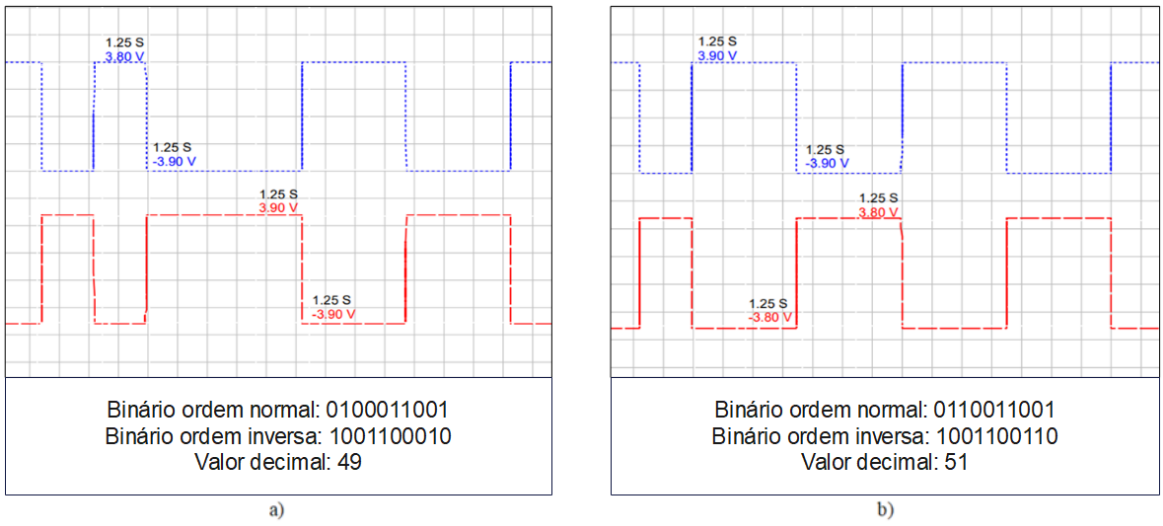
solicitação de temperatura, é uma função GET). A Figura 49-b e a Figura 50-a apresentam os sinais referentes aos bytes 3 e 4, que armazenam o valor da temperatura em si. Por fim, a Figura 50-b e a Figura 51 apresentam os bytes 5 e 6, os quais representam o CRC para que seja feita a verificação de erros na transmissão das mensagens.

Figura 48 - Tensão RS-485 em resposta à requisição de solicitação de temperatura: a) Byte 0; b) Byte 1;



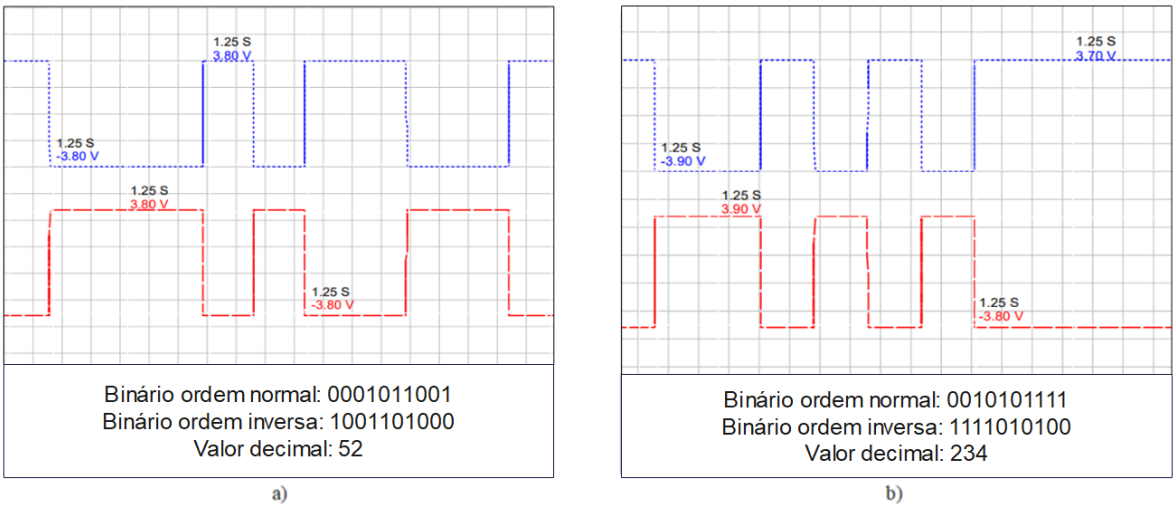
Fonte: Autoria Própria

Figura 49 - Tensão RS-485 em resposta à requisição de solicitação de temperatura: a) Byte 2; b) Byte 3;



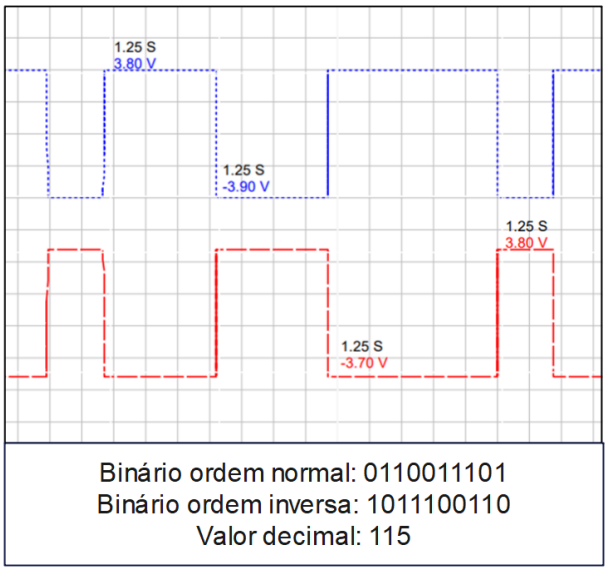
Fonte: Autoria Própria

Figura 50 - Tensão RS-485 em resposta à requisição de solicitação de temperatura: a) Byte 4; b) Byte 5;



Fonte: Autoria Própria

Figura 51 - Tensão RS-485 em resposta à requisição de solicitação de temperatura: Byte 6



Fonte: Autoria Própria

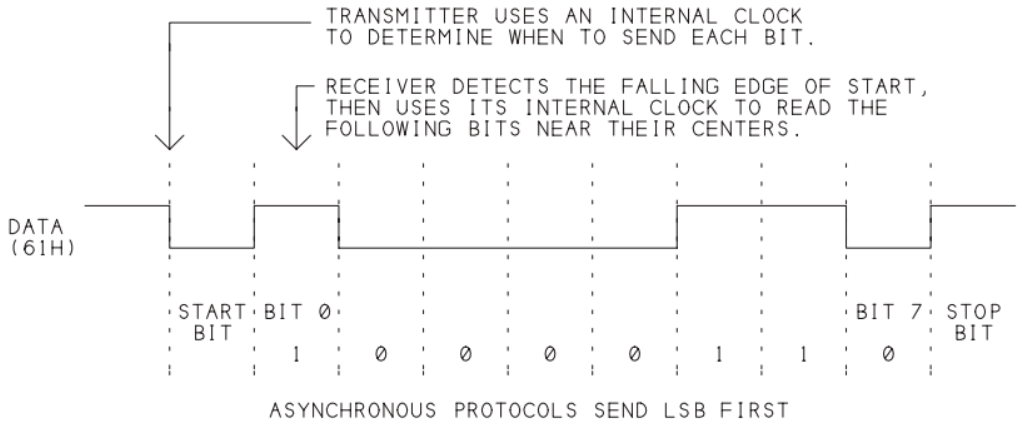
A partir da análise da leitura da tensão das linhas A e B para cada byte avaliado, pode ser visto que o valor lógico resultante corresponde ao esperado (de acordo a Figura 32). Para os sinais de tensão avaliados, é observado que a tensão na linha A é sempre contrária à da linha B, como esperado no protocolo RS-485. Além disso, pode ser visto que o nível de tensão mais alto para ambas as linhas é de 3,9 V, enquanto o mais baixo é de -3,9 V, fazendo com que um nível lógico alto seja representado pela tensão de aproximadamente 7,8 V e o

nível lógico baixo por uma tensão de aproximadamente -7,8 V. Esses valores de tensão estão condizentes com o esperado para o protocolo empregado.

Visando deixar explícito os valores lógicos representados pelos sinais lidos, foram colocados nas imagens os bits obtidos a partir da avaliação da tensão $V_A - V_B$. É importante notar que os bits constituintes de cada byte devem ser lidos da direita para a esquerda para correta avaliação e interpretação do caractere equivalente em ASC-II. Além disso, nota-se que cada onda representativa de um byte apresenta na verdade 10 bits e não 8, fato decorrente da comunicação serial assíncrona estabelecida.

Como apresentado por Axelson (2007), a comunicação serial de um computador tem como componente chave a UART (*Universal Asynchronous Receiver Transmitter*), pegando bytes de dados em paralelo e transmitindo-os de forma sequencial. Essa estrutura de dados formada a partir da UART é chamada de "palavra", a qual tipicamente possui um *start bit* (igual a 0), 8 bits de dados (1 byte), um bit de paridade (opcional, não foi utilizado no projeto) e um *stop bit* (igual a 1). Além disso, é característico nessa transmissão o envio do bit menos significativo primeiro, tornando necessário interpretar os dados no sentido contrário. Esse *frame* descrito é dado por um protocolo de mais baixo nível, o qual pode ser visto na Figura 52.

Figura 52 - Transmissão de dados assíncrona



Fonte: Axelson (2007)

Essas características definidas para a transmissão do sinal, a partir da UART, explicam a questão dos dois bits a mais encontrados nas leituras dos sinais, assim como o sentido de leitura dos dados. É interessante notar que enquanto foi criado um protocolo de alto nível,

cujo *frame* pode ser observado na Figura 28, existe um protocolo de baixo nível responsável por encapsular cada byte de dados.

Avaliando todos os sinais e interpretando corretamente o valor lógico inerente a cada um, pode ser visto que a comunicação para a solicitação de temperatura foi feita com sucesso. Isso evidencia a efetividade da estrutura criada, desde a organização dos dados pelo servidor (enviando os bytes adequados corretamente), até o recebimento pelo arduino (o qual interpreta a mensagem e organiza uma resposta para ser enviada de volta ao servidor). Além disso, tem-se também a efetividade da parte física da comunicação, evidenciando que não ocorreram problemas na transmissão da informação e que todos os tipos de sinais trabalhados foram convertidos adequadamente a partir dos circuitos integrados empregados.

O exemplo avaliado foi para uma solicitação de temperatura ao dispositivo de campo localizado no ambiente com *id* 11, sendo retornado o valor de 34 ºC. A análise do nível de tensão para uma solicitação de atualização de *setpoint* ocorre de forma análoga, sendo somente alterado o tipo de informação contida em alguns dos bytes. Para o projeto desenvolvido foram abordadas somente as operações de atualização de *setpoint* e de requisição de temperatura, no entanto, diversas outras requisições podem ser estabelecidas à medida que o projeto é expandido, uma vez definida a estrutura básica para que o processo ocorra.

5 CONCLUSÃO

A partir do desenvolvimento do presente projeto foi possível a criação de uma API REST a qual permitiu a automação de um processo de controle e monitoramento de temperatura residencial. A API permitiu uma integração completa entre os usuários que utilizam o serviço, um banco de dados (o qual realiza o armazenamento de informações atualizadas e relevantes em relação a cada ambiente) e os dispositivos de campo que atuam nos ambientes (os quais estabelecem o gerenciamento local dos dispositivos de refrigeração/aquecimento).

O projeto desenvolvido criou uma estrutura básica com potencial de expansão em diversos aspectos. A partir dessa estrutura, as requisições e operações definidas podem ser feitas não somente a nível local, mas também à longas distâncias, a partir do momento que é realizada a conexão via internet. Desse modo, qualquer usuário com acesso permitido consegue ter o controle dos ambientes no qual o sistema atua.

Outro ponto é que o estudo de caso realizado para controle de temperatura foi apenas um exemplo. Os microcontroladores empregados podem estabelecer controle e monitoramento de saídas e entradas diversas. Um sensor de umidade, por exemplo, poderia ser conectado ao microcontrolador, permitindo ao usuário saber o nível de umidade no ambiente. Para isso devem ser feitas algumas alterações envolvendo a base de dados, o tratamento de dados realizado pelo servidor, além da criação de uma nova rota de requisição. No entanto, o ponto principal é que a estrutura física que permite a comunicação entre o servidor e os microcontroladores alocados nos ambientes já está feita, o que viabiliza a troca de qualquer tipo de informação.

Um outro ponto que evidencia a capacidade de expansão do projeto é o próprio protocolo RS-485 empregado, o qual permite que até 256 microcontroladores sejam conectados ao barramento comum existente entre esses e o servidor. Além disso, existe a questão do comprimento máximo elevado dos cabos (aproximadamente 1219,2 m), permitindo conexões longas entre o microcontrolador e o servidor, ou seja, áreas grandes podem ser monitoradas facilmente.

REFERÊNCIAS

ALDRICH, Frances. Smart homes: past, present and future. *In:* HARPER; Richard. **Inside the smart home**. London: Springer London, 2003. p. 17-39.

ALONSO, Gustavo; CASATI, Fabio; KUNO, Harumi; MACHIRAJU, Vijay. **Web Services:** concepts, architectures and applications. Heidelberg: Springer Berlin, 2004.

ALVES, William. **Banco de dados**. São Paulo: Érica, 2014.

AMADEU, Claudia. **Banco de dados**. São Paulo: Pearson Education do Brasil, 2015.

API Web Services beginner tutorial 1 - introduction - what is a Web Service. *[S. l.: s. n]*, 2016. 1 vídeo (09:28 min). Publicado pelo canal Automation Step by Step. Disponível em: https://www.youtube.com/watch?v=oTzNRv6X51o&t=11s. Acesso em: 16 dez. 2023.

API Web Services beginner tutorial 2 – how Web Services work (overview). *[S. l.: s. n]*, 2016. 1 vídeo (05:26 min). Publicado pelo canal Automation Step by Step. Disponível em: https://www.youtube.com/watch?v=ktl57lfw-68&t=1s. Acesso em: 16 dez. 2023.

API Web Services beginner tutorial 3 – what is WSDL and UDDI. *[S. l.: s. n]*, 2016. 1 vídeo (05:00 min). Publicado pelo canal Automation Step by Step. Disponível em: https://www.youtube.com/watch?v=MUq_RkG7De0&t=17s. Acesso em: 16 dez. 2023.

API Web Services beginner tutorial 4 – what are SOAP Web Services. *[S. l.: s. n]*, 2016. 1 vídeo (06:50 min). Publicado pelo canal Automation Step by Step. Disponível em: https://www.youtube.com/watch?v=sTGgBoFBDAY. Acesso em: 16 dez. 2023.

API Web Services beginner tutorial 5 – what are REST Web Services (part-1). *[S. l.: s. n]*, 2016. 1 vídeo (12:47 min). Publicado pelo canal Automation Step by Step. Disponível em: https://www.youtube.com/watch?v=lyxJQWUymV4. Acesso em: 16 dez. 2023.

API Web Services beginner tutorial 6 – what are REST Web Services (part-2). *[S. l.: s. n]*, 2016. 1 vídeo (08:11 min). Publicado pelo canal Automation Step by Step. Disponível em: https://www.youtube.com/watch?v=YvKlUy7mDNU. Acesso em: 16 dez. 2023.

API Web Services beginner tutorial 7 –how to create API documentation through WSDL URL. *[S. l.: s. n]*, 2016. 1 vídeo (02:49 min). Publicado pelo canal Automation Step by Step. Disponível em: https://www.youtube.com/watch?v=EaWAX_Lu9Bw. Acesso em: 16 dez. 2023.

AXELSON, Jan. **Serial port complete**: COM ports, USB virtual COM ports, and ports for embedded systems. 2nd ed. Madison: Lakeview Research, 2007.

BEIRL, Diana; ROGERS, Yvone; YUILL, Nicola. Using voice assistant skills in family life. *In:* INTERNATIONAL CONFERENCE ON COMPUTER SUPPORTED COLLABORATIVE LEARNING, 13., 2019, Lião. **Proceedings**[…]. Londres: ULC Discovery, 2019. p. 96-103. Disponível em: https://discovery.ucl.ac.uk/id/eprint/10084820/. Acesso em:16 dez. 2023.

DOUGLAS, Korry; DOUGLAS, Susan. **PostgreSQL**: a comprehensive guide to building, programming, and administering PostgreSQL databases. 2nd ed. Carmel: Sams Publishing, 2005.

ELIAS, Gledson; LOBATO, Luis Carlos. **Arquitetura e protocolos de rede TCP-IP**. 2. ed. Rio de Janeiro: Escola Superior de Redes, 2013. Disponível em: http://www.lest5.com.br/lest/attachments/article/3/ArquiteturaeProtocolosdeRedeTCPIP.pdf. Acesso em: 16 dez. 2023.

FERNÁNDEZ, Marcial. **Rede de computadores**. 2. ed. Fortaleza: EdUECE, 2015. Disponível em: https://educapes.capes.gov.br/bitstream/capes/432642/2/Livro%20%20Redes%20de%20Computadores.pdf. Acesso em: 16 dez. 2023.

FORTUNE BUSINESS INSIGHTS. **Smart home market**. [2023]. 1 fotografia, color. 629 kB. Formato PNG. Disponível em: https://www.fortunebusinessinsights.com/infographics/smart-home-market.png. Acesso em: 21 jan. 2024.

GOEKING, Weruska. Da máquina a vapor aos softwares de automação. **O Setor Elétrico**, São Paulo, n. 52, p. 70-77, maio 2010. Disponível em: https://www.voltimum.com.br/sites/www.voltimum.com.br/files/memoria_maio_10.pdf. Acesso em: 16 dez. 2023.

GO in 100 seconds. *[S.l.: s.n]*, 2022. 1 vídeo (2:29 min). Publicado pelo canal Fireship. Disponível em: https://www.youtube.com/watch?v=446E-r0rXHI. Acesso em: 03 jan. 2024.

GOLANG (a linguagem do futuro?) // dicionário do programador. *[S.l: s.n]*, 2020. 1 vídeo (8:53 min). Publicado pelo canal Código Fonte TV. Disponível em: https://www.youtube.com/watch?v=2kyNEf9IsBQ. Acesso em: 03 jan. 2024.

GO. **Chi**. São Francisco: Google, [2023]. Disponível em: https://pkg.go.dev/github.com/go-chi/chi. Acesso em: 16 dez. 2023.

GO. **Go for web development**. São Francisco: Google, [2019]. Disponível em: https://go.dev/solutions/webdev. Acesso em: 16 dez. 2023.

HERCOG, Drago; **Communication protocols**: principles, methods and specifications. Cham: Springer International Publishing, 2020.

IBM. **Web services overview**. Nova Iorque: IBM Corporation, [2021]. Disponível em: https://www.ibm.com/docs/en/rsas/7.5.0?topic=applications-web-services-overview. Acesso em: 16 dez. 2023.

JUBA, Salahaldin; VANNAHME, Achin; VOLKOV, Andrey. **Learning PostgreSQL**. Birmingham: Packt Publishing, 2015. Disponível em: https://www.google.com.br/books/edition/Learning_PostgreSQL/jfKoCwAAQBAJ?hl=pt-BR&gbpv=0. Acesso em: 16 dez. 2023.

KUMAR, Sumit; DALAL, Sumit; DIXIT, Vivek. The OSI model: overview on the seven layers of computer networks. **International Journal of Computer Science and Information Technology Research**, v. 2, n. 3, p. 461-466, July/Sep. 2014. Disponível em: https://www.researchpublish.com/upload/book/THE%20OSI%20MODEL%20OVERVIEW%20ON%20THE%20SEVEN%20LAYERS-607.pdf. Acesso em: 02 nov. 2023.

KUROSE, James; ROSS, Keith. **Redes de computadores e a internet:** uma abordagem top-down. 8. ed. São Paulo: Grupo A, 2021. Disponível em: https://plataforma.bvirtual.com.br. Acesso em: 02 nov. 2023.

LABCENTER ELECTRONICS. **Proteus**. Versão 8. North Yorkshire: Labcenter Electronics, c2013.

MEDEIROS, Luciano. **Banco de dados:** princípios e prática. Curitiba: Intersaberes, 2013.

NERY, Amanda; ROCHA, Jessé; GUAZZI, Érika. Estudos sobre os códigos SPC e CRC. *In:* ENCONTRO REGIONAL DE MATÉMATICA APLICADA E COMPUTACIONAL DO RIO GRANDE DO SUL, 10., 2020, Pelotas. **Anais** [...]. Rio Grande do Sul: Editora PUCRS, 2020. Disponível em: https://editora.pucrs.br/edipucrs/acessolivre/anais/1501/assets/edicoes/2020/arquivos/39.pdf. Acesso em: 02 dez 2023.

TANENBAUM, Andrew; FEAMSTER, Nick; WETHERALL, David. **Redes de computadores**. 6. ed. Porto Alegre: Bookman, 2021.

TOSCHI, Guilherme; CAMPOS, Leonardo; CUGNASCA, Carlos. Home automation networks: a survey. **Computer standards e interfaces**, São Paulo, v. 50, p. 42-54, 2017. DOI: https://doi.org/10.1016/j.csi.2016.08.008. Disponível em: https://www.sciencedirect.com/science/article/pii/S0920548916300654?via%3Dihub. Acesso em: 16 dez. 2023.

WANG, Kankan; LIU, Hao. **Technical white paper: RS-485 basic series**. Texas: Texas Instruments, [2021]. Disponível em: https://www.ti.com/lit/wp/slla545/slla545.pdf?ts=1704275906992&ref_url=https%253A%252F%252Fwww.google.fr%252F. Acesso em: 02 nov. 2023.

WEG. **Modbus RTU SRW 01**: manual do usuário. 2014. Disponível em: https://static.weg.net/medias/downloadcenter/ha7/h11/WEG-srw01-manual-da-comunicacao-modbus-10000013033-6.0x-manual-portugues-br.pdf. Acesso em: 03 jan. 2024.

WORTMEYER, Charles; FREITAS, Fernando; CARDOSO, Líuam. Automação residencial: busca de tecnologias visando o conforto, a economia, a praticidade e a segurança do usuário. *In:* SIMPÓSIO DE EXCELÊNCIA EM GESTÃO E TECNOLOGIA, 2., 2005, Rio de Janeiro. **Anais** [...]. Rio de Janeiro: Associação Educacional Dom Bosco, 2005. Disponível em: https://www.aedb.br/seget/arquivos/artigos05/256_SEGET%20-%20Automacao%20Residencial.pdf. Acesso em: 18 jan. 2024.

APÊNDICE A – main.go

```
package main

import (
    "database/sql"
    "fmt"
    "log"
    "net/http"
    "github.com/go-chi/chi"
    "github.com/go-chi/cors"
    _ "github.com/lib/pq"
)

func main() {

    channelOne := make(chan bool)
    channelTwo := make(chan bool)
    var db *sql.DB
    var err error

    db, err = sql.Open(PostgresDriver, DataSourceName)

    if err != nil {
        panic(err.Error())
    } else {
        fmt.Println("Connected!")
    }

    config := setSerialConfig("COM5", 9600)
    fmt.Println(config)
    port := openSerialPort(config)

    router := chi.NewRouter()

    router.Use(cors.Handler(cors.Options{
        AllowedOrigins: []string{"https://*", "http://*"},
        AllowedMethods: []string{"GET", "POST", "PATCH", "DELETE"},
        AllowedHeaders: []string{"*"},
        ExposedHeaders: []string{"Link"},
        MaxAge:         300,
    }))

    go getAllTemps(port, db, channelOne, channelTwo)
    router.Get("/DadosAmbientes", handlerGetAllAmb(db))
    router.Get("/DadosAmbientes/{ambienteID}", handlerGetSpecificAmb(db))
    router.Get("/DadosAmbientes/DadosDevices", handlerGetAllDevices(db))
```

```
    router.Post("/AdicionarAmbiente", handlerPostAmb(db, port, channelOne, channelTwo))
    router.Delete("/DeletarAmbiente/{ambienteID}", handlerDeletAmb(db))
    router.Patch("/AtualizarSetpoint/{ambienteID}", handlerUpdateSetpoint(db, port,
channelOne, channelTwo))

    server := &http.Server{
        Handler: router,
        Addr:    ":8080",
    }

    log.Printf("Server starting on port 8080")

    err = server.ListenAndServe()

    if err != nil {
        log.Fatal(err)
    }
}
```

APÊNDICE B – json.go

```go
package main

import (
    "encoding/json"
    "log"
    "net/http"
)

func respondWithError(w http.ResponseWriter, code int, msg string) {

    type errResponse struct {
        Error string `json:"error"`
    }
    respondWithJSON(w, code, errResponse{
        Error: msg,
    })
}

func respondWithJSON(w http.ResponseWriter, code int, payload interface{}) {

    dat, err := json.Marshal(payload)

    if err != nil {
        log.Printf("Failed to marshal JSON response: %v", payload)
        w.WriteHeader(500)
        return
    }

    w.Header().Add("Content-Type", "application/json")
    w.WriteHeader(code)
    w.Write(dat)
}

func readingJSON(r *http.Request) (map[string]any, error) {

    request := map[string]any{}
    err := json.NewDecoder(r.Body).Decode(&request)

    if err != nil {
        log.Printf("Failed to unmarshal JSON response: %v", r.Body)
        return nil, err
    }

    return request, nil
}
```

APÊNDICE C – handler_postAmb.go

```
package main

import (
    "database/sql"
    "fmt"
    "net/http"
    "time"

    "github.com/tarm/serial"
)

func handlerPostAmb(db *sql.DB, port *serial.Port, channelOne, channelTwo chan bool)
http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

        fmt.Println("")
        fmt.Printf("Post Ambient Request! \n")
        fmt.Printf("\n")

        getAllTemps := true
        channelOne <- true

        go func() {
            for {
                getAllTemps = <-channelTwo
            }
        }()

        request, _ := readingJSON(r)

        if request["nome"] == "" {
            respondWithError(w, 400, "To insert a new ambient, it must be a valid name!")
            fmt.Println("To insert a new ambient, it must be inserted a valid name!")
            fmt.Printf("\n")
            channelOne <- false
            return
        }

        _, ok := request["setpoint"].(float64)

        if !ok {
            respondWithError(w, 400, "To insert a new ambient, it must be a valid
setpoint!")
            fmt.Println("To insert a new ambient, it must be inserted a valid setpoint!")
            fmt.Printf("\n")
```

```
            channelOne <- false
            return
        }

        _, ok = request["nome"].(string)

        if !ok {
            respondWithError(w, 400, "To insert a new ambient, it must be a valid name!")
            fmt.Println("To insert a new ambient, it must be inserted a valid name!")
            fmt.Printf("\n")
            channelOne <- false
            return
        }

        intSetpoint := int(request["setpoint"].(float64))

        intSetPointInitValue := intSetpoint

        if intSetpoint < setpointMinValue || intSetpoint > setpointMaxValue {
            intSetpoint = 20
        }

        ambient := Ambient{
            Nome:     request["nome"].(string),
            SetPoint: intSetpoint,
            ID:       int(request["id"].(float64)),
        }

        err := sqlInsert(db, ambient)

        if err != nil {
            respondWithError(w, 500, "Error to register new ambient! Problem updating
database!")
            fmt.Println("Error to register new ambient! Problem updating database!")
            fmt.Printf("\n")
            channelOne <- false
            return
        }

        for {
            if !getAllTemps {

                fmt.Printf("\n")
                fmt.Printf("New ambient is being registered! \n")
                fmt.Printf("\n")

                time.Sleep(4 * time.Second)
```

```
                fmt.Printf("Message sent to field device in order to inform setpoint
value = %v \n", updateSetpointSendData(ambient.ID, intSetpoint))
                fmt.Println("Waiting for field device response!")
                err = sendData(port, updateSetpointSendData(ambient.ID, intSetpoint))

                if err != nil {
                    respondWithError(w, 500, "Error to update setpoint! Problem to send
data to field device.")
                    fmt.Println("Error to update setpoint! Problem to send data to field
device!")
                    fmt.Printf("\n")
                    channelOne <- false
                    return
                }

                updateSetpointReceivedData, err := receiveData(port)

                fmt.Printf("Received message from field device: %v \n",
updateSetpointReceivedData)

                if err != nil {
                    respondWithError(w, 500, "Error to update setpoint! Problem receiving
data from field device.")
                    fmt.Println("Error to update setpoint! Problem receiving data from
field device!")
                    fmt.Printf("\n")
                    channelOne <- false
                    return
                }

                crcReceivedBytes := calculateCRC16(updateSetpointReceivedData)

                if (crcReceivedBytes[0] == updateSetpointReceivedData[5]) &&
(crcReceivedBytes[1] == updateSetpointReceivedData[6]) {

                    fmt.Println("CRC Valid!")
                    updateSetpointSuccess_0 := string(updateSetpointReceivedData[3])
                    updateSetpointSuccess 1 := string(updateSetpointReceivedData[4])

                    if updateSetpointSuccess_0 != "1" || updateSetpointSuccess_1 != "1" {
                        respondWithError(w, 500, "Error to update setpoint! Confirmation
bytes content (bytes 3 and 4) received from field device are not the expected!")
                        fmt.Println("Error to update Setpoint! Confirmation bytes content
(bytes 3 and 4) received from field device are not the expected!")
                        fmt.Printf("\n")
                        channelOne <- false
                        return
                    }
```

```
                } else {
                    respondWithError(w, 500, "Error to update Setpoint! Confirmation
message from field device is  corrupted (CRC mismatch)!")
                    fmt.Println("Error to update Setpoint! Received confirmation message
from field Device was corrupted (CRC mismatch)!")
                    fmt.Printf("\n")
                    channelOne <- false
                    return
                }

                if (intSetPointInitValue < setpointMinValue) || (intSetPointInitValue >
setpointMaxValue) {
                    string := fmt.Sprintf("Temperature of %v is out of apropriate range,
default setpoint value (%v) was set. Ambient registered succesfully!",
intSetPointInitValue, setpointdefaultValue)
                    respondWithJSON(w, 201, string)
                    fmt.Printf("Temperature of %v is out of appropriate range, default
setpoint value (%v) was set. Ambient registered succesfully!", intSetPointInitValue,
setpointdefaultValue)
                    fmt.Printf("\n")
                } else {
                    respondWithJSON(w, 201, "New ambient registered succesfully!")
                    fmt.Println("New ambient registered succesfully!")
                    fmt.Printf("\n")
                }

                time.Sleep(12 * time.Second)
                channelOne <- false
                break

            } else {
                continue
            }
        }
    }
}
```

APÊNDICE D – handler_updateSetpoint.go

```
package main

import (
    "database/sql"
    "fmt"
    "net/http"
    "strconv"
    "time"

    "github.com/go-chi/chi"
    "github.com/tarm/serial"
)

func updateSetpointSendData(ID, setpoint int) (updateSetpointSendDataByte []byte) {

    IDString := strconv.Itoa(ID)
    setpointString := strconv.Itoa(setpoint)
    updateSetpointSendDataByte = append(updateSetpointSendDataByte, []byte(IDString)...)
    updateSetpointSendDataByte = append(updateSetpointSendDataByte, []byte("2")...)
    updateSetpointSendDataByte = append(updateSetpointSendDataByte,
[]byte(setpointString)...)
    crcBytes := calculateCRC16(updateSetpointSendDataByte)
    updateSetpointSendDataByte = append(updateSetpointSendDataByte, crcBytes...)

    return updateSetpointSendDataByte
}

func handlerUpdateSetpoint(db *sql.DB, port *serial.Port, channelOne, channelTwo chan
bool) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

        fmt.Printf("\n")
        fmt.Printf("Update Setpoint Request! \n")
        fmt.Printf("\n")

        getAllTemps := true
        channelOne <- true

        go func() {
            for {
                getAllTemps = <-channelTwo
            }
        }()

        requestedAmbientID, _ := strconv.Atoi(chi.URLParam(r, "ambienteID"))
```

```
        allAmbients, _ := sqlSelectAll(db)

        request, _ := readingJSON(r)

        _, ok := request["setpoint"].(float64)

        if !ok {
            respondWithError(w, 400, "To update setpoint, it must be a valid setpoint!")
            fmt.Println("To update setpoint, it must be a valid setpoint!")
            fmt.Printf("\n")
            channelOne <- false
            return
        }

        intSetpoint := int(request["setpoint"].(float64))
        intSetPointInitValue := intSetpoint

        if intSetpoint < setpointMinValue || intSetpoint > setpointMaxValue {
            intSetpoint = 20
        }

        err := sqlUpdateSetpoint(db, requestedAmbientID, intSetpoint)

        if err != nil {
            respondWithError(w, 500, "Error do update setpoint! Problem updating
database.")
            fmt.Println("Error to update setpoint! Problem with database!")
            fmt.Printf("\n")
            channelOne <- false
            return
        }

        for i := 1; i <= sqlElementCount(db); i++ {
            if allAmbients.Ambients[i-1].ID == requestedAmbientID {
                break
            } else if i == sqlElementCount(db) {
                respondWithError(w, 400, "ID informed is not associated with any ambient
registered!")
                fmt.Println("ID informed is not associated with any ambient registered!")
                fmt.Printf("\n")
                channelOne <- false
                return
            }
        }

        for {
            if !getAllTemps {

                fmt.Printf("\n")
```

```
                fmt.Printf("Setpoint is being updated! \n")
                fmt.Printf("\n")

                time.Sleep(4 * time.Second)

                fmt.Printf("Message sent to field device in order to update setpoint
value = %v \n", updateSetpointSendData(requestedAmbientID, intSetpoint))
                fmt.Println("Waiting for field device response!")
                err = sendData(port, updateSetpointSendData(requestedAmbientID,
intSetpoint))

                if err != nil {
                    respondWithError(w, 500, "Error do update setpoint! Problem to send
data to field device.")
                    fmt.Println("Error to update setpoint! Problem to send data to field
device!")
                    fmt.Printf("\n")
                    channelOne <- false
                    return
                }

                updateSetpointReceivedData, err := receiveData(port)

                fmt.Printf("Received message from field device: %v \n",
updateSetpointReceivedData)

                if err != nil {
                    respondWithError(w, 500, "Error do update setpoint! Problem receiving
data from field device.")
                    fmt.Println("Error to update setpoint! Problem receiving data from
field device!")
                    fmt.Printf("\n")
                    channelOne <- false
                    return
                }

                crcReceivedBytes := calculateCRC16(updateSetpointReceivedData)

                if (crcReceivedBytes[0] == updateSetpointReceivedData[5]) &&
(crcReceivedBytes[1] == updateSetpointReceivedData[6]) {

                    fmt.Println("CRC Valid!")

                    updateSetpointSuccess_0 := string(updateSetpointReceivedData[3])
                    updateSetpointSuccess_1 := string(updateSetpointReceivedData[4])

                    if updateSetpointSuccess_0 != "1" || updateSetpointSuccess_1 != "1" {
                        respondWithError(w, 500, "Error to update setpoint! Confirmation
bytes content (bytes 3 and 4) received from field device are not the expected!")
```

```
                        fmt.Println("Error to update Setpoint! Confirmation bytes content
(bytes 3 and 4) received from field device are not the expected!")
                        fmt.Printf("\n")
                        channelOne <- false
                        return
                    }

                } else {
                    respondWithError(w, 500, "Error to update Setpoint! Received
confirmation message from field Device was corrupted (CRC mismatch)!")
                    fmt.Println("Error to update setpoint! Confirmation message from
field device is corrupted (CRC mismatch)!")
                    fmt.Printf("\n")
                    channelOne <- false
                    return
                }

                if (intSetPointInitValue < setpointMinValue) || (intSetPointInitValue >
setpointMaxValue) {
                    requestResponse := fmt.Sprintf("Temperature of %v is out of
appropriate range, default setpoint value (%v) was set", intSetPointInitValue,
setpointdefaultValue)
                    fmt.Printf("Temperature of (%v) is out of appropriate range, default
setpoint value of %v °C was set. Ambient setpoint succesfully updated!",
intSetPointInitValue, setpointdefaultValue)
                    fmt.Printf("\n")
                    respondWithJSON(w, 201, requestResponse)
                } else {
                    respondWithJSON(w, 201, "Ambient setpoint succesfully updated!")
                    fmt.Println("Ambient setpoint succesfully updated!")
                    fmt.Printf("\n")
                }

                time.Sleep(12 * time.Second)
                channelOne <- false
                break

            } else {
                continue
            }
        }
    }
}
```

APÊNDICE E – handler_getSpecificAmbient.go

```
package main

import (
    "database/sql"
    "fmt"
    "net/http"
    "strconv"
    "github.com/go-chi/chi"
)

func handlerGetSpecificAmb(db *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

        fmt.Printf("\n")
        fmt.Println("Get Specific Ambient was requested!")
        fmt.Printf("\n")
        requestedAmbientID, _ := strconv.Atoi(chi.URLParam(r, "ambienteID"))
        allAmbients,   := sqlSelectAll(db)

        for i := 1; i <= sqlElementCount(db); i++ {
            if allAmbients.Ambients[i-1].ID == requestedAmbientID {
                break
            } else if i == sqlElementCount(db) {
                respondWithError(w, 400, "ID informed is not associated with any
registered ambient.")
                fmt.Println("ID informed is not associated with any ambient registered!")
                fmt.Printf("\n")
                return
            }
        }

        ambientData, err := sqlSelectByID(db, requestedAmbientID)

        if err == nil {
            respondWithJSON(w, 200, ambientData)
            fmt.Println("Ambient data was returned!")
            fmt.Printf("\n")
        } else {
            respondWithError(w, 500, "Error to return registered ambient data! Problem
returning data from database.")
            fmt.Println("Error to return registered ambient data! Problem with database")
            fmt.Printf("\n")
        }
    }
}
```

APÊNDICE F – handler_getAllDevices.go

```
package main

import (
    "database/sql"
    "fmt"
    "net/http"
)

func handlerGetAllDevices(db *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

        fmt.Printf("\n")
        fmt.Println("Get All Devices was requested!")
        fmt.Printf("\n")

        allDevices, err := sqlSelectAllDevices(db)

        if err == nil {
            respondWithJSON(w, 200, allDevices)
            fmt.Println("All devices data were returned!")
            fmt.Printf("\n")
        } else {
            respondWithError(w, 500, "Error to return all devices data! Problem returning
data from database.")
            fmt.Println("Error to return all devices data! Problem with database.")
            fmt.Printf("\n")
        }

    }
}
```

APÊNDICE G – handler_getAllAmb.go

```
package main

import (
    "database/sql"
    "fmt"
    "net/http"
)

func handlerGetAllAmb(db *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

        fmt.Printf("\n")
        fmt.Println("Get All Ambients was requested!")
        fmt.Printf("\n")

        allAmbients, err := sqlSelectAll(db)

        if err == nil {
            respondWithJSON(w, 200, allAmbients)
            fmt.Println("All registered ambients data were returned!")
            fmt.Printf("\n")
        } else {
            respondWithError(w, 500, "Error to return all registered ambient data!
Problem returning data from database.")
            fmt.Println("Error to return all registered ambients data! Problem with
database!")
            fmt.Printf("\n")
        }

    }
}
```

APÊNDICE H – handler_deletAmb.go

```
package main

import (
    "database/sql"
    "fmt"
    "net/http"
    "strconv"
    "github.com/go-chi/chi"
)

func handlerDeletAmb(db *sql.DB) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {

        fmt.Printf("\n")
        fmt.Println("Ambient Removal Request!")
        fmt.Printf("\n")
        requestedAmbientID, _ := strconv.Atoi(chi.URLParam(r, "ambienteID"))
        allAmbients,   := sqlSelectAll(db)

        for i := 1; i <= sqlElementCount(db); i++ {
            if allAmbients.Ambients[i-1].ID == requestedAmbientID {
                break
            } else if i == sqlElementCount(db) {
                respondWithError(w, 400, "ID informed is not associated with any
registered ambient.")
                fmt.Println("ID informed is not associated with any ambient registered!")
                fmt.Printf("\n")
                return
            }
        }

        err := sqlDelete(db, requestedAmbientID)

        if err == nil {
            respondWithJSON(w, 200, "Ambient deleted succesfully!")
            fmt.Println("Ambient deleted succesfully!")
            fmt.Printf("\n")
        } else {
            respondWithError(w, 500, "Error to delete ambient! Problem to delete on
database.")
            fmt.Println("Error to delete ambient! Problem with database.")
            fmt.Printf("\n")
        }
    }
}
```

APÊNDICE I – getaAllTemps_serial.go

```
package main

import (
    "database/sql"
    "fmt"
    "strconv"
    "time"

    "github.com/tarm/serial"
)

func getAllTempsSendData(AllAmbients AllAmbients, i int) (getAllTempsSendDataByte []byte)
{

    ID := strconv.Itoa(AllAmbients.Ambients[i].ID)

    getAllTempsSendDataByte = append(getAllTempsSendDataByte, []byte(ID)...)
    getAllTempsSendDataByte = append(getAllTempsSendDataByte, []byte("1")...)
    getAllTempsSendDataByte = append(getAllTempsSendDataByte, []byte("0")...)
    getAllTempsSendDataByte = append(getAllTempsSendDataByte, []byte("0")...)

    crcBytes := calculateCRC16(getAllTempsSendDataByte)

    getAllTempsSendDataByte = append(getAllTempsSendDataByte, crcBytes...)

    return getAllTempsSendDataByte
}

func getAllTemps(port *serial.Port, db *sql.DB, channelOne, channelTwo chan bool) {

    handlerUpdateSetpoint := false

    go func() {
        for {
            handlerUpdateSetpoint = <-channelOne
        }
    }()

    for {
        if !handlerUpdateSetpoint {

            time.Sleep(100 * time.Millisecond)
            fmt.Println("")
            fmt.Println(">> Get All Ambients Temperature Routine!")
```

```
            allAmbients, err := sqlSelectAll(db)

            if err == nil {
                for i := 0; i < sqlElementCount(db); i++ {

                    leftBothLoops := false

                    for i := 0; i < 20; i++ {
                        if handlerUpdateSetpoint {
                            leftBothLoops = true
                            break
                        }
                        time.Sleep(1 * time.Second)
                    }

                    if leftBothLoops {
                        break
                    }

                    fmt.Println("")
                    fmt.Printf(">> Monitoring temperature of ambient with id = %v \n",
allAmbients.Ambients[i].ID)
                    sendData(port, getAllTempsSendData(allAmbients, i))
                    fmt.Printf(">> Message sent to field device requesting temperature
value: %v \n", getAllTempsSendData(allAmbients, i))
                    fmt.Println(">> Waiting for field device response!")
                    tempsReceivedSlice, err := receiveData(port)

                    if err != nil {
                        fmt.Println(err)
                        continue
                    }

                    fmt.Printf(">> Received message with temperature content: %v \n",
tempsReceivedSlice)

                    crcReceivedBytes := calculateCRC16(tempsReceivedSlice)

                    if (crcReceivedBytes[0] == tempsReceivedSlice[5]) &&
(crcReceivedBytes[1] == tempsReceivedSlice[6]) {

                        fmt.Println(">> CRC Valid!")
                        decTemperatureString := string(tempsReceivedSlice[3])
                        uniTemperatureString := string(tempsReceivedSlice[4])
                        decTemperatureInt, _ := strconv.Atoi(decTemperatureString)
                        uniTemperatureInt, _ := strconv.Atoi(uniTemperatureString)
                        err = sqlUpdateTemperature(db, allAmbients.Ambients[i].ID,
(decTemperatureInt*10)+uniTemperatureInt)
```

```
                        fmt.Printf(">> Ambient Temperature (ambient_id = %v) was updated
succesfully! \n", allAmbients.Ambients[i].ID)

                        if err != nil {
                            fmt.Println(">> Error to update temperatude in database!")
                        }

                    } else {
                        fmt.Println(">> CRC Invalid!")
                    }

                    if handlerUpdateSetpoint {
                        break
                    }
                }

            } else {
                fmt.Println(">> Error to update temperature")
            }

        } else {
            channelTwo <- false
            continue
        }

        time.Sleep(10 * time.Second)
    }
}
```

APÊNDICE J – serial_funcs.go

```
package main

import (
    "errors"
    "log"
    "time"
    "github.com/tarm/serial"
)

const (
    polynomial uint16 = 0x8005
)

func setSerialConfig(Name string, Baud int) (config *serial.Config) {

    config = &serial.Config{
        Name:        Name,
        Baud:        Baud,
        ReadTimeout: 1000,
    }
    return config
}

func openSerialPort(config *serial.Config) (port *serial.Port) {

    port, err := serial.OpenPort(config)
    if err != nil {
        log.Fatal(err)
    }
    return port
}

func calculateCRC16(data []byte) []byte {

    crcBytes := make([]byte, 2)
    var crc uint16 = 0xFFFF

    for b := 0; b < 5; b++ {
        crc ^= uint16(data[b])
        for i := 0; i < 8; i++ {
            if crc&1 == 1 {
                crc = (crc >> 1) ^ polynomial
            } else {
                crc >>= 1
            }
```

```
        }
    }

    crcBytes[0] = byte(crc & 0xFF)
    crcBytes[1] = byte((crc >> 8) & 0xFF)

    return crcBytes
}

func sendData(port *serial.Port, dataToSend []byte) error {

    port.Flush()
    _, err := port.Write(dataToSend)

    if err != nil {
        log.Fatal(err)
    }

    return err
}

func receiveData(port *serial.Port) ([]byte, error) {

    n := 0
    nTotal := 0
    init := 0
    var err error
    buffer := make([]byte, 7)
    var bufferFull []byte
    bufferCopy := make([]byte, 7)
    count := 0

    for {
        for i := 0; i < 7; i++ {
            buffer[i] = 0
        }

        n = 0
        n, err = port.Read(buffer)

        for _, b := range buffer {
            if b != 0 {
                init++
                break
            }
        }

        falseCount := 0
```

```
        for i := 0; i < 7; i++ {
            if (buffer[i] != bufferCopy[i]) || (init == 0) {
                falseCount++
            }
        }

        if falseCount == 0 {
            break
        } else {
            if buffer[1] != 0 {
                bufferFull = append(bufferFull, buffer...)
            }
        }

        nTotal = nTotal + n
        copy(bufferCopy, buffer)
        time.Sleep(2 * time.Second)

        if count >= 20 {
            return []byte{}, errors.New("no result received")
        }

        count++
    }

    return bufferFull, err
}
```

APÊNDICE K – SQL_funcs.go

```
package main

import (
    "database/sql"
    "fmt"
)

func sqlSelectAll(db *sql.DB) (AllAmbients AllAmbients, err error) {

    sqlStatement, err := db.Query("SELECT name, temperature, setpoint, ambient_id FROM "
+ TableAmbientes)

    for sqlStatement.Next() {

        var ambiente Ambient
        err = sqlStatement.Scan(&ambiente.Nome, &ambiente.Temperatura,
&ambiente.SetPoint, &ambiente.ID)

        AllAmbients.AddAmbient(Ambient{
            Nome:        ambiente.Nome,
            Temperatura: ambiente.Temperatura,
            SetPoint:    ambiente.SetPoint,
            ID:          ambiente.ID,
        })
    }

    return AllAmbients, err
}

func sqlSelectByID(db *sql.DB, ID int) (ambiente Ambient, err error) {

    sqlStatement := fmt.Sprintf("SELECT name, temperature, setpoint, ambient_id FROM %s
where ambient_id = $1", TableAmbientes)
    query := db.QueryRow(sqlStatement, ID)
    err = query.Scan(&ambiente.Nome, &ambiente.Temperatura, &ambiente.SetPoint,
&ambiente.ID)

    return ambiente, err
}

func sqlSelectAllDevices(db *sql.DB) (allDevices AllDevices, err error) {

    sqlStatement, err := db.Query("SELECT device_id, device_model, power_btu, ambient_id
FROM " + TableDevices)
```

```
    for sqlStatement.Next() {

        var device Device
        err = sqlStatement.Scan(&device.Device_ID, &device.Modelo, &device.PotenciaBTU,
&device.Ambient_ID)

        allDevices.AddDevice(Device{
            Device_ID:   device.Device_ID,
            Modelo:      device.Modelo,
            PotenciaBTU: device.PotenciaBTU,
            Ambient_ID:  device.Ambient_ID,
        })
    }

    return allDevices, err
}

func sqlElementCount(db *sql.DB) (count int) {

    _ = db.QueryRow("SELECT COUNT(*) FROM ambiente").Scan(&count)

    return count
}

func sqlInsert(db *sql.DB, ambiente Ambient) error {

    sqlStatement := fmt.Sprintf("INSERT INTO %s VALUES ($1,$2,$3,$4)", TableAmbientes)
    insert, _ := db.Prepare(sqlStatement)
    _, err := insert.Exec(ambiente.Nome, 0, ambiente.SetPoint, ambiente.ID)

    return err
}

func sqlUpdateSetpoint(db *sql.DB, ID int, setpoint int) error {

    sqlStatement := fmt.Sprintf("update %s set setpoint=$1 where ambient_id=$2",
TableAmbientes)
    update,   := db.Prepare(sqlStatement)
    _, err := update.Exec(setpoint, ID)

    return err
}

func sqlUpdateTemperature(db *sql.DB, ID int, temperatura int) error {

    sqlStatement := fmt.Sprintf("update %s set temperature=$1 where ambient_id=$2",
TableAmbientes)
    update, _ := db.Prepare(sqlStatement)
    _, err := update.Exec(temperatura, ID)
```

```
    return err
}

func sqlDelete(db *sql.DB, ID int) error {

    sqlStatement := fmt.Sprintf("delete from %s where ambient_id=$1", TableAmbientes)
    delete, _ := db.Prepare(sqlStatement)
    _, err := delete.Exec(ID)

    return err
}
```

APÊNDICE L – config.go

```go
package main

import (
    "fmt"
)

type Ambient struct {
    Nome        string `json:"nome"`
    Temperatura int    `json:"temperatura"`
    SetPoint    int    `json:"setpoint"`
    ID          int    `json:"id"`
}

type Device struct {
    Modelo      string `json:"modelo"`
    PotenciaBTU int    `json:"potencia_btu"`
    Device_ID   int    `json:"device_id"`
    Ambient ID  int    `json:"ambient id"`
}

type AllDevices struct {
    Devices []Device
}

type AllAmbients struct {
    Ambients []Ambient
}

func (t *AllAmbients) AddAmbient(ambient Ambient) {
    t.Ambients = append(t.Ambients, ambient)
}

func (t *AllDevices) AddDevice(device Device) {
    t.Devices = append(t.Devices, device)
}

const PostgresDriver = "postgres"

const User = "postgres"

const Host = "localhost"

const Port = "5432"

const Password = "TESTE1234"
```

```
const DbName = "ambients"

const TableAmbientes = "ambiente"

const TableDevices = "devices"

var DataSourceName = fmt.Sprintf("host=%s port=%s user=%s "+
    "password=%s dbname=%s sslmode=disable", Host, Port, User, Password, DbName)

const setpointMinValue = 5

const setpointMaxValue = 50

const setpointdefaultValue = 20
```

APÊNDICE M – CÓDIGO IMPLEMENTADO NOS MICROCONTROLADORES

```
#include <Arduino.h>
#include <LiquidCrystal.h>

#define DeRe_Max_487    12
#define led_stpt     10
#define led_temp     11

const uint16_t polynomial = 0x8005;
char id[] = "11";
char GET = '1';
char PATCH = '2';
char setpoint[] = "20";

LiquidCrystal lcd(2,3,4,5,6,7);

uint16_t calculateCRC16(const uint8_t *data, size_t length) {
  uint16_t crc = 0xFFFF;

  for (size_t i = 0; i < length; i++) {
    crc ^= data[i];
    for (int j = 0; j < 8; j++) {
      if (crc & 1) {
        crc = (crc >> 1) ^ 0x8005;
      } else {
        crc >>= 1;
      }
    }
  }

  return crc;
}

void uint8ArrayToString(uint8_t *array, size_t length, char *output) {
  for (size_t i = 0; i < length; i++) {
    output[i] = (char)array[i];
  }
  output[length] = '\0';
}

void limparArray(char array[], int tamanho) {
  for (int i = 0; i < tamanho; i++) {
    array[i] = '\0';
  }
}
```

```
void limparBufferSerial() {
  while (Serial.available() > 0) {
    char dadoDescartado = Serial.read();
  }
}

void setup() {
  Serial.begin(9600);
  pinMode(DeRe_Max_487, OUTPUT);
  pinMode(led_temp_11, OUTPUT);
  pinMode(led_stpt_11, OUTPUT);
  digitalWrite(DeRe_Max_487, LOW);
  digitalWrite(led_temp_11, LOW);
  digitalWrite(led_stpt_11, LOW);

  lcd.begin(16,2);
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Display iniciado");
}

void loop() {

  int tempAnalogico = analogRead(A1);
  int tempMapeado = map(tempAnalogico, 0, 1023, 5, 40);

  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Aguardando");
  lcd.setCursor(0,1);
  lcd.print(Serial.available());
  delay(200);

  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print("Temp:");
  lcd.setCursor(7,0);
  lcd.print(tempMapeado);
  lcd.setCursor(0,1);
  lcd.print("Setpt:");
  lcd.setCursor(7,1);
  lcd.print(setpoint);
  delay(200);

  if (Serial.available() > 0) {
    size_t messageLength = 7;

    uint8_t message[messageLength];
    Serial.readBytes(message, messageLength);
```

```
    uint16_t calculatedCRC = calculateCRC16(message, 5);

    if (message[5] == (calculatedCRC & 0xFF) && message[6] == ((calculatedCRC >> 8) &
0xFF)) {
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("CRC CERTO");
      delay(100);
    } else {
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("CRC ERRADO");
      delay(100);
    }

    if ((char(message[0]) != id[0]) || (char(message[1]) != id[1])){
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("ID N/ Corresponde");
      delay(500);

      if(char(message[2]) == GET){
        delay(4000);

        limparBufferSerial();
      } else if(char(message[2]) == PATCH){
        delay(3400);

        limparBufferSerial();
      }

    } else {
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("ID Corresponde");
      delay(500);

      if(char(message[2]) == GET){
      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("Enviando");
      lcd.setCursor(0,1);
      lcd.print("Temperatura");

      digitalWrite(led_temp, HIGH);

      uint8_t respostaSolicitacaoGET[7];
```

```
      respostaSolicitacaoGET[0] = uint8_t(id[0]);
      respostaSolicitacaoGET[1] = uint8_t(id[1]);
      respostaSolicitacaoGET[2] = uint8_t(GET);

      tempAnalogico = analogRead(A1);
      tempMapeado = map(tempAnalogico, 0, 1023, 5, 40);

      char tempMapeadoArray[2];

      itoa(tempMapeado, tempMapeadoArray, 10);

      respostaSolicitacaoGET[3] = uint8_t(tempMapeadoArray[0]);

      respostaSolicitacaoGET[4] = uint8_t(tempMapeadoArray[1]);

      uint16_t CRCrespostaSolicitacaoGET = calculateCRC16(respostaSolicitacaoGET, 5);

      respostaSolicitacaoGET[5] = CRCrespostaSolicitacaoGET & 0xFF;

      respostaSolicitacaoGET[6] = (CRCrespostaSolicitacaoGET >> 8) & 0xFF;

      digitalWrite(DeRe_Max_487, HIGH);

      delay(500);

      Serial.write(respostaSolicitacaoGET, 7);

      delay(500);

      digitalWrite(DeRe_Max_487, LOW);

      delay(3000);

      digitalWrite(led_temp, LOW);

      limparBufferSerial();

    } else if(char(message[2]) == PATCH){

      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("Atualizando");
      lcd.setCursor(0,1);
      lcd.print("Setpoint");

      setpoint[0] = char(message[3]);
      setpoint[1] = char(message[4]);

      digitalWrite(led_stpt, HIGH);
```

```
      uint8_t respostaSolicitacaoPATCH[7];

      respostaSolicitacaoPATCH[0] = uint8_t(id[0]);
      respostaSolicitacaoPATCH[1] = uint8_t(id[1]);
      respostaSolicitacaoPATCH[2] = uint8_t(PATCH);

      respostaSolicitacaoPATCH[3] = uint8_t('1');
      respostaSolicitacaoPATCH[4] = uint8_t('1');

      lcd.clear();
      lcd.setCursor(0,0);
      lcd.print("Enviando");
      lcd.setCursor(0,1);
      lcd.print("Confirmacao");

      uint16_t CRCrespostaSolicitacaoPATCH = calculateCRC16(respostaSolicitacaoPATCH, 5);

      respostaSolicitacaoPATCH[5] = CRCrespostaSolicitacaoPATCH & 0xFF;

      respostaSolicitacaoPATCH[6] = (CRCrespostaSolicitacaoPATCH >> 8) & 0xFF;

      digitalWrite(DeRe_Max_487, HIGH);

      delay(500);

      Serial.write(respostaSolicitacaoPATCH, 7);

      delay(500);

      digitalWrite(DeRe_Max_487, LOW);

      delay(3000);

      digitalWrite(led_stpt, LOW);

      limparBufferSerial();

      lcd.clear();
    }
    }
    limparArray(message, 7);
  }
}
```