Trabalho final de Teleinformática de Redes 1

Universidade de Brasília
Departamento de Ciências da Computação

Autores:
Henrique Givisiez dos Santos (Matrícula: 21/1027563)
Gabriel Francisco de Oliveira Castro (Matrícula: 20/2066571)
André Rodrigues Modesto (Matrícula: 21/1068324)

Professor:
Marcelo Marotta

Brasília, Distrito Federal
Julho de 2025

I. Visão Geral

Este relatório tem como propósito apresentar a simulação do funcionamento das camadas de enlace e física de uma comunicação digital, por meio da implementação de protocolos fundamentais. Nesse sentido, serão abordados, de forma estruturada e individualizada, os seguintes processos: enquadramento de dados, modulação em banda base e modulação por portadora. Para cada um desses tópicos, será feita uma análise detalhada tanto da perspectiva do transmissor quanto do receptor, destacando os procedimentos realizados em cada etapa da comunicação, bem como o modo que foi implementado no código. Vale ressaltar que, para cada tópico que inclui implementação em código, o nome do aluno responsável será indicado no início da respectiva subseção.

Para facilitar a compreensão e visualização, todo o sistema será integrado em um único arquivo e apresentado através de uma interface gráfica interativa. Essa interface permitirá acompanhar e visualizar em tempo real a transformação dos dados nas diferentes fases da comunicação, tornando os conceitos teóricos mais acessíveis e intuitivos.

II. Camada Física

Primeiramente, a camada física representa a base para toda comunicação de dados. Nessa perspectiva, a camada física é responsável por receber o trem de bits, já formatados pela camada de enlace, e o converte em um sinal elétrico ou magnetico que possa ser transmitido por um meio físico. No recepetor, esta camada realiza o processo inverso, ou seja, amostra o sinal recebido para decodifica-lo de volta à sequência original de bits.

Conforme o enunciado do projeto, a implementação da camada física foi dividida em duas subetas principais: a modulação digital e a modulação por portadora.

A. Modulação Digital - Desenvolvido por: Henrique Givisiez

Do ponto de vista geral, a modulação digital é o processo responsável por converter a sequência de bits (0s e 1s) em um sinal digital para a transmissão. Nesse contexto, a escolha do esquema de codificação é crucial, haja vista que afeta características importantes do sinal, como a sincronização entre transmissor e receptor, bem como a presença de uma componenente de corrente contínua (DC).

Para esse projeto, foram implementados três esquemas de modulação digital que serão abordados a seguir. Cabe ressaltar que para testar as modulações, foi utilizado o trem de bits da figura 1:

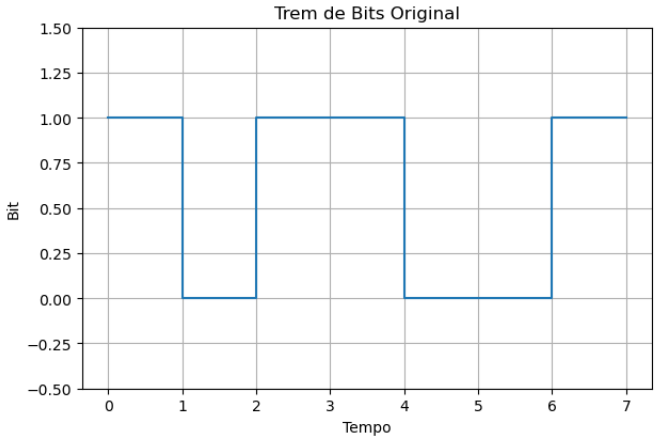

Figura 1. Trem de bits para modulações digitais.

a) Non-Return-to-Zero Polar (NRZ-Polar)

É uma técnica que busca balancear o sinal utilizando dois níveis distintos de voltagem: $-V$ para representar o bit 0 e $+V$ para o bit 1. Do ponto de vista do receptor, a reconstrução do trem de bits é feita comparando a voltagem recebida com os níveis esperados, permitindo identificar corretamente cada bit transmitido.

```
19      def modular(self, bits: list[int]) -> list[int]:
20          """
21          A modulacao parte do transmissor que irá  convertes os dados (bits) em um sinal do meio para o receptor poder receber
22
23          Args:
24              bits (list[int]): Dados(bits) que irao ser convertidos em uma funcao continua (voltagem V)
25          Returns:
26              list[int]: +V para bit 1 e -V para bit 0
27          """
28          return [1 if bit == 1 else -1 for bit in bits]
29
30      def demodular(self, sinais: list[int]) -> list[int]:
31          """
32          A demodulacao parte do receptor que irá receber as variacoes de voltagem e interpretar em bits (1 para +V e 0 para -V)
33
34          Args:
35              bits (list[int]): Sinais (variacoes da amplitude) da voltagem
36
37          Returns:
38              list[int]: Sequencia de bits (1 para +V e 0 para -V)
39          """
40          return [1 if s > 0 else 0 for s in sinais]
```

Figura 2. Parte do script utilizada para implementação da modulação digital NRZ-Polar.

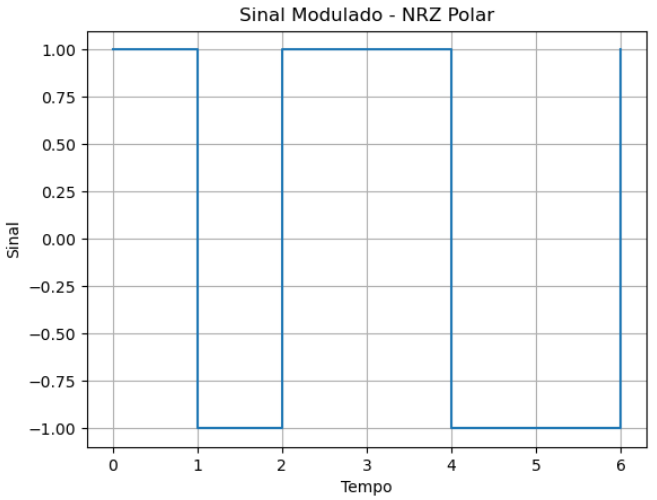

Figura 3. Sinal modulado por NRZ-Polar.

b) Manchester

Tem como objetivo garantir a sincronização entre transmissor e receptor, utilizando uma função boolena exclusiva OR (XOR) para combinar os sinais de clock e dados em um único trem de bits. Nesse sentido, cada período de bit reflete a transição de um nível de tensão para outro, bem como a transição sempre ocorre no ponto médio do período de bit, fornecendo uma indicação clara do estado do bit.

```python
def modular(self, bits: list[int]) -> list[int]:
    """
    Modula os bits usando a codificacao Manchester.
    Cada bit eh representado por dois sinais, onde:
    - 0 eh representado por [0, 1]
    - 1 eh representado por [1, 0]

    Args:
        bits (list[int]): Lista de bits a serem modulados (0s e 1s).

    Returns:
        list[int]: Lista de sinais modulados, onde cada bit eh representado por dois sinais.
    Examples:
        Entrada: [0, 1, 1, 0]
        Saida: [0, 1, 1, 0, 1, 0, 0, 1]
    """
    sinais = []
    clock = [0, 1] # Sinal de clock para a codificacao Manchester
    # Cada bit eh convertido em dois sinais
    for bit in bits:
        sinais.append(bit ^ clock[0])
        sinais.append(bit ^ clock[1])
    return sinais

def demodular(self, sinais: list[int]) -> list[int]:
    """
    Demodula os sinais usando a codificacao Manchester.
    Cada par de sinais representa um bit, onde:
    - [0, 1] eh interpretado como 0
    - [1, 0] eh interpretado como 1

    Args:
        sinais (list[int]): Lista de sinais a serem demodulados, onde cada bit eh representado por dois sinais.

    Returns:
        list[int]: Lista de bits demodulados (0s e 1s).
    Examples:
        Entrada: [0, 1, 1, 0, 1, 0, 0, 1]
        Saida: [0, 1, 1, 0]
    """
    # Lê dois sinais por bit
    bits = []
    for i in range(0, len(sinais), 2):
        if sinais[i] < sinais[i + 1]:
            bits.append(0)  # Transição de baixo para cima
        else:
            bits.append(1)  # Transição de cima para baixo
    return bits
```

Figura 4. Parte do script utilizada para implementação da modulação digital Manchester.

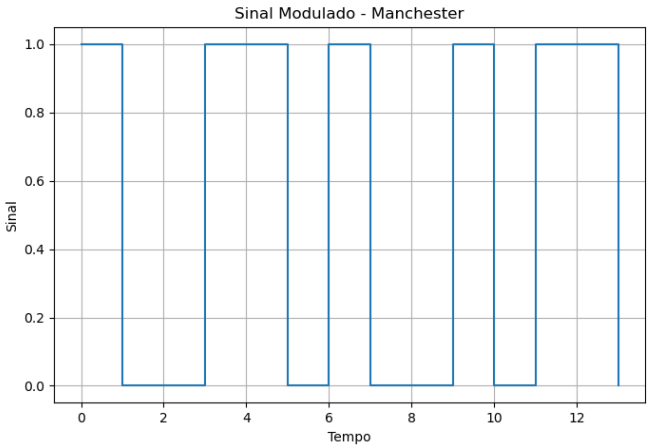

Figura 5. Sinal modulado por Manchester.

c) Bipolar

O esquema de codificação bipolar utiliza três níveis de tensão: positivo, negativo e zero. O bit 0 é representado por zero volts, enquanto o bit 1 é representado alternadamente por tensões positivas e negativas. Do ponto de vista do receptor, a demodulação é feita analisando os níveis de tensão recebidos: quando o sinal é zero, interpreta-se como bit 0; quando o sinal apresenta uma tensão diferente de zero, seja positiva ou negativa, interpreta-se como bit 1. A alternância esperada entre os pulsos positivos e negativos também serve como mecanismo de verificação e ajuda a manter a sincronização.

```python
def modular(self, bits: list[int]) -> list[int]:
    """
    Modula uma sequencia de bits usando codificacao bipolar.

    Args:
        bits (list[int]): lista de bits (0s e 1s) a serem modulados.

    Returns:
        list[int]: lista de niveis de tensao para transmissao:
            0 -> 0V
            1 -> alterna +1 e -1 a cada 1 sucessivo

    Examples:
        Entrada: [1, 0, 1, 1, 0, 1]
        Saida:   [+1, 0, -1, +1, 0, -1]
    """
    sinais = []
    ultimo = -1  # Comeca em -1 para que o primeiro 1 gere +1
    for bit in bits:
        if bit == 0:
            sinais.append(0)
        else:
            ultimo *= -1  # alterna entre +1 e -1
            sinais.append(ultimo)
    return sinais

def demodular(self, sinais: list[int]) -> list[int]:
    """
    Demodula uma sequencia de sinais de volta para bits usando codificao bipolar.

    Args:
        sinais (list[int]): lista de niveis de tensao recebidos.

    Returns:
        list[int]: lista de bits reconstruidos:
                    - 0V → 0
                    - +V ou -V → 1 (nao se verifica violacao aqui)

    Examples:
        Entrada: [+1, 0, -1, +1, 0, -1]
        Saida:   [1, 0, 1, 1, 0, 1]
    """
    return [0 if s == 0 else 1 for s in sinais]
```

Figura 6. Parte do script utilizada para implementação da modulação digital Bipolar.

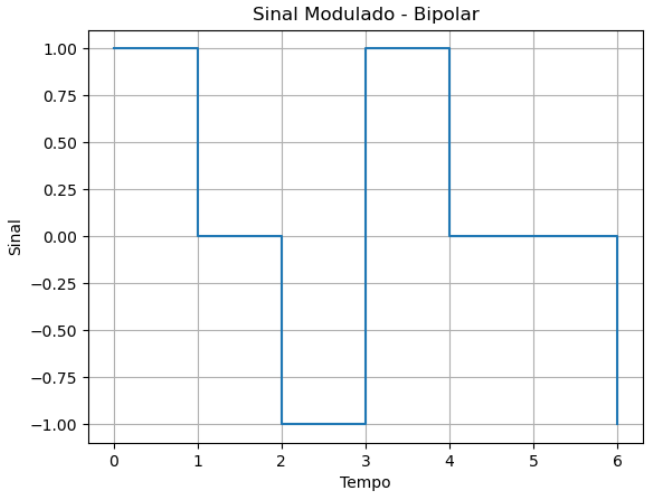

Figura 7. Sinal modulado por Bipolar.

B. Modulação por Portadora - Desenvolvido por: Gabriel Castro

De maneira simplificada, a modulação por portadora é uma técnica utilizada para transmitir sinais digitais por meio da variação de uma onda senoidal de alta frequência, chamada de portadora. Nessa perspectiva, o principal objetivo é adaptar o sinal de dados para que ele possa ser transmitido de forma eficiente e confiável em diferentes meios físicos Nesse tipo de modulação, as características da portadora — como amplitude, frequência ou fase — são alteradas conforme os bits a serem transmitidos. Dessa forma, diferentes técnicas podem ser utilizadas dependendo das necessidades do sistema. Do ponto de vista do receptor, a demodulação consiste em identificar essas

variações na portadora para reconstruir o sinal digital original. Esse processo pode exigir sincronização com a portadora e o uso de filtros ou detectores específicos, dependendo da técnica empregada.

a) Amplitude Shift Keying (ASK)

É uma técnica em que a amplitude da portadora varia conforme os bits transmitidos, mantendo fase e frequência constantes, o bit 1 é representado pela presença da portadora e o bit 0 pela sua ausência. Abaixo, apresenta-se uma saída exemplo do código, bem como uma explicação breve do para facilitar o entendimento.

```python
    def modular(self, bits: list[int]) -> np.ndarray:
        """
        Converte uma sequência de bits em um sinal ASK. Para isso, "multiplica" a portadora
        pelo modulante digital
        """
        sinal_modulado = []
        for bit in bits:
            if bit == 1:
                amplitude = 1.0
            else:
                amplitude = 0
            sinal_modulado.extend(amplitude * self.portadora)

        return np.array(sinal_modulado)

    def demodular(self, sinais: np.ndarray) -> list[int]:
        """
        O princípio da demodulação ASK é medir a energia do sinal em cada intervalo (Este método recebe a onda
        ASK e o converte para o sinal original").
        - Criamos um valor de corte para entre 0 e 1, isso leva em consideração o mundo real e o ruído.
        - Dessa forma, calculamos a energia do trecho (soma dos quadrados das amostras) e compara com o limiar
        p/ decidir qual o valor do bit.
        """
        bits_recuperados = []
        for i in range(0, len(sinais),self.amostras_por_bit):
            trecho = sinais[i:i + self.amostras_por_bit]
            energia = np.sum(trecho**2)/self.amostras_por_bit
            if energia > self.limiar_de_energia:
                bits_recuperados.append(1)
            else:
                bits_recuperados.append(0)

        return bits_recuperados
```

Figura 8. Parte do script utilizada para implementação da modulação por portadora ASK.

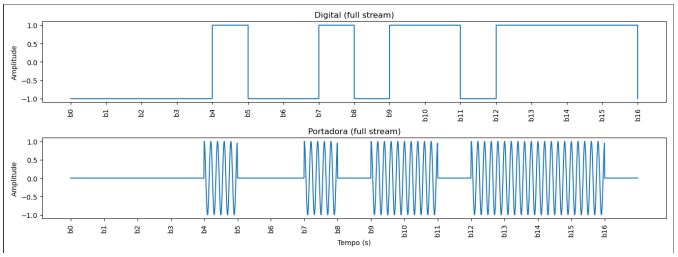

Figura 9. Modulação ASK para a mensagem ”o”.

Este trecho de código implementa a modulação ASK (Amplitude Shift Keying). Ele recebe uma sequência de bits e, para cada bit, gera um segmento de sinal. Se o bit for 1, o sinal de portadora é enviado com sua amplitude máxima (1); se o bit for 0, a amplitude é zero, resultando em ausência de sinal. Esses segmentos são concatenados para formar o sinal modulado final.

b) Frequency Shift Keying (FSK)

A Modulação por Chaveamento de Frequência (FSK) é uma técnica de modulação digital que representa dados variando a frequência de um sinal portador. Essencialmente, ela associa diferentes frequências a bits específicos do sinal modulante. Por exemplo, um bit ’1’ pode corresponder a uma frequência de portadora (fp_1), enquanto um bit ’0’ pode ser representado por outra frequência (fp_2). Abaixo, apresenta-se uma saída exemplo do código, bem como uma explicação breve do para facilitar o entendimento.

Este trecho de código implementa a modulação FSK (Frequency Shift Keying). Ele recebe uma sequência de bits e, para cada bit, seleciona uma frequência de portadora diferente. Se o bit for 1, ele estende o sinal com uma portadora com determinada frequência, enquanto que se o bit for 0, ele usa outra portadora com uma frequência distinta. O resultado é um sinal modulado onde a informação é transmitida pela mudança entre duas frequências.

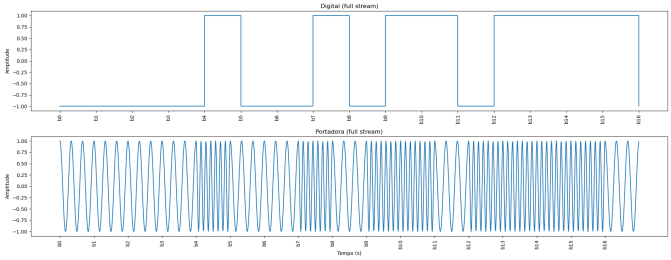

Figura 10. Modulação FSK para o caracter ”o”.

```python
    def modular(self, bits: list[int]) -> np.ndarray:
        sinal_modulado = []
        for bit in bits:
            if bit == 1:
                sinal_modulado.extend(self.portadora_1)
            else:
                sinal_modulado.extend(self.portadora_0)

        return np.array(sinal_modulado)

    def demodular(self, sinal: np.ndarray) -> list[int]:
        """
        Converte um sinal FSK de volta para uma sequência de bits. Para isso< Compara cada trecho do sinal com
        duas ondas de referência (para bit 0 e 1) e escolhe o bit correspondente à referência mais parecida.
        """
        bits_recuperados = []
        for i in range(0, len(sinal), self.amostras_por_bit):
            trecho = sinal[i:i + self.amostras_por_bit]
            correlacao_0 = np.sum(trecho * self.portadora_0)
            correlacao_1 = np.sum(trecho * self.portadora_1)
            if correlacao_1 > correlacao_0:
                bits_recuperados.append(1)
            else:
                bits_recuperados.append(0)

        return bits_recuperados
```

Figura 11. Parte do script utilizada para implementação da modulação por portadora FSK.

c) 8-Quadrature Amplitude Modulation (8-QAM)

É uma técnica de modulação digital que representa dados combinando variações na amplitude e na fase de uma onda portadora. Em 8-QAM, grupos de três bits são mapeados para símbolos únicos, o que significa que cada símbolo transmite 3 bits de informação ($2^3 = 8$ símbolos distintos). Durante a modulação, os bits de entrada são agrupados em trios, e cada trio é associado a um ponto específico na constelação 8-QAM, que define a amplitude e a fase a serem transmitidas. Abaixo, apresenta-se uma saída exemplo do código, bem como uma explicação breve do para facilitar o entendimento.

```python
    def modular(self, bits: list[int]) -> np.ndarray:
        """
        Modula uma lista de bits (múltiplos de 3) em um sinal 8-QAM.

        Parâmetros:
        - bits: Lista de bits (0 ou 1). Ex: [1, 0, 1, 0, 0, 0].

        Retorna:
        - Sinal modulado em formato numpy.array.

        Lança:
        - ValueError se o número de bits não for múltiplo de 3.
        """
        if len(bits) % 3 != 0:
            raise ValueError("A quantidade de bits deve ser múltipla de 3 (8-QAM usa 3 bits/símbolo).")

        sinal_modulado = np.array([])
        for i in range(0, len(bits), 3):
            simbolo = (bits[i] << 2) | (bits[i+1] << 1) | bits[i+2]
            amp, fase = self._constelacao[simbolo]
            simbolo_modulado = amp * np.cos(2 * np.pi * self.freq_portadora * self._tempo + fase)
            sinal_modulado = np.append(sinal_modulado, simbolo_modulado)

        return sinal_modulado

    def demodular(self, sinal: np.ndarray) -> list[int]:
        """Demodula o sinal 8-QAM para bits."""
        if len(sinal) % self.amostras_por_simbolo != 0:
            raise ValueError("O comprimento do sinal deve ser múltiplo de amostras_por_simbolo.")

        bits_recuperados = []
        for i in range(0, len(sinal), self.amostras_por_simbolo):
            trecho = sinal[i:i + self.amostras_por_simbolo]
            i_rx = (2 / self.amostras_por_simbolo) * np.sum(trecho * self._portadora_i)
            q_rx = (2 / self.amostras_por_simbolo) * np.sum(trecho * self._portadora_q)
            simbolo_detectado = min(
                self._iq_ideal.keys(),
                key=lambda s: (i_rx - self._iq_ideal[s][0])**2 + (q_rx - self._iq_ideal[s][1])**2
            )
            bits_recuperados.extend([
                (simbolo_detectado >> 2) & 1,
                (simbolo_detectado >> 1) & 1,
                simbolo_detectado & 1
            ])

        return bits_recuperados
```

Figura 12. Parte do script utilizada para implementação da modulação por portadora 8-QAM.

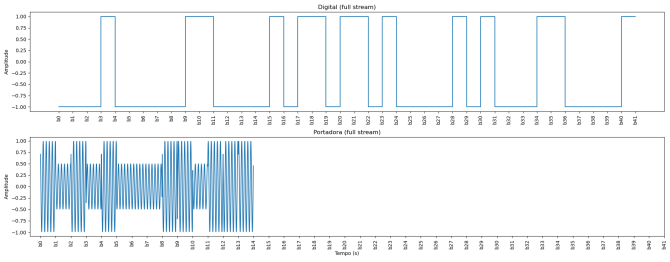

Figura 13. Modulação 8-QAM para a mensagem ”ala”.

Este código realiza a modulação 8-QAM. Ele agrupa os bits de entrada em blocos de 3, pois cada bloco representa um símbolo QAM. Para cada grupo de 3 bits, o código determina a amplitude e fase correspondentes a partir de uma constelação predefinida. Em seguida, ele gera o sinal modulado para aquele símbolo, combinando a amplitude e a fase com uma portadora de frequência específica ao longo do tempo. O resultado é um sinal modulado complexo, construído pela concatenação desses símbolos modulados.

III. Camada de Enlace

De forma geral e simplificada, a Camada de Enlace desempenha um papel essencial na comunicação de dados, ao transformar o fluxo bruto de bits da Camada Física em unidades organizadas e confiáveis, chamadas quadros (frames), que podem ser interpretadas corretamente pelas camadas superiores. Neste trabalho, será abordado três etapas fundamentais dessa camada. Primeiro, serão implementados protocolos de enquadramento de dados, responsáveis por delimitar o início e o fim de cada unidade de informação transmitida, utilizando as técnicas de contagem de caracteres, enquadramento com FLAGS e inserção de bytes ou caracteres, e enquadramento com FLAGS e inserção de bits. Em seguida, serão tratados os protocolos de detecção de erros, que verificam a integridade dos dados por meio do bit de paridade par e do código CRC com polinômio CRC-32, conforme o padrão IEEE 802. Por fim, será adicionado o protocolo de correção de erros utilizando o Código de Hamming, que permite identificar e corrigir automaticamente erros simples durante a transmissão. A seguir, cada um desses tópicos será detalhado separadamente.

A. Protocolo de Enquadramento de Dados - Desenvolvido por: André Modesto

1) Contagem de caracteres

De maneira simplificada, utiliza um campo no cabeçalho para especificar o numero de bytes no quadrado, ou seja, o cabeçalho específica o tamanho do quadro.

```python
class ContagemCaracteres(Enquadrador):
    """
    Enquadramento por contagem de caracteres.
    Adiciona um cabeçalho de 8 bits com o tamanho do quadro.
    """

    def enquadrar(self, bits: list[int]) -> list[int]:
        tamanho = len(bits)
        tamanho_bits = list(map(int, f"{tamanho:08b}"))  # converte tamanho para 8 bits
        return tamanho_bits + bits

    def desenquadrar(self, quadro: list[int]) -> list[int]:
        tamanho = int("".join(map(str, quadro[:8])), 2)  # extrai tamanho
        return quadro[8:8 + tamanho]

```

Figura 14. Parte do script utilizada para implementação do protocolo de enquadramento de dados utilizando contagem de caracacteres.

2) Enquadramento com FLAGS e inserção de bytes ou caracteres

Para evitar que sequências de bits dentro dos dados sejam erroneamente interpretadas como marcadores de fim de quadro (bytes FLAG), este método utiliza um byte especial de FLAG para delimitar o início e o fim de cada quadro. Caso o byte FLAG ocorra nos dados, um byte de escape é inserido antes dele. Dois bytes FLAG consecutivos indicam o final de um quadro e o início do seguinte.

```python
    def enquadrar(self, bits: list[int]) -> list[int]:
        bytes_ = self.bits_to_bytes(bits)
        stuffed_bytes = []

        for b in bytes_:
            if b == 0x7E:  # FLAG
                stuffed_bytes.extend([0x7D, 0x5E])  # ESC + substituto
            elif b == 0x7D:  # ESC
                stuffed_bytes.extend([0x7D, 0x5D])  # ESC + substituto
            else:
                stuffed_bytes.append(b)

        stuffed_bits = []
        for b in stuffed_bytes:
            stuffed_bits.extend(self.byte_to_bits(b))

        return self.FLAG + stuffed_bits + self.FLAG

    def desenquadrar(self, quadro: list[int]) -> list[int]:
        dados = quadro[8:-8]  # remove FLAGs
        bytes_ = self.bits_to_bytes(dados)
        i = 0
        result = []
        while i < len(bytes_):
            if bytes_[i] == 0x7D:  # ESC
                i += 1
                if i < len(bytes_):
                    if bytes_[i] == 0x5E:
                        result.append(0x7E)
                    elif bytes_[i] == 0x5D:
                        result.append(0x7D)
            else:
                result.append(bytes_[i])
            i += 1

        # Converte lista de bytes de volta para bits
        return sum([self.byte_to_bits(b) for b in result], [])

```

Figura 15. Parte do script utilizada para implementação do protocolo de enquadramento de dados utilizando enquadramento com FLAGS e inserção de bytes ou caracteres.

O trecho acima converte a mensagem em bytes e, para evitar que bytes de dados se confundam com delimitadores (FLAG ou ESC), insere um byte de escape (0x7D) antes de cada ocorrência desses caracteres especiais, modificando-os. A mensagem assim ”recheada”é então delimitada por FLAGs no início e fim. No desenquadramento, o processo é invertido: as FLAGs são removidas e os bytes de escape são localizados e removidos, restaurando os bytes originais da mensagem para sua forma de bits.

3) Enquadramento com FLAGS e Inserção de bits

Similar à inserção de bytes, este método opera em nível de bits. Quadros podem ter tamanhos arbitrários, pois o enquadramento é realizado através da inserção de bits. Cada quadro inicia e termina com uma sequência de bits específica, o padrão de FLAG 01111110 (ou 0x7E em hexadecimal). A seguir, segue trecho de código a fim de detalhar a implementação e facilitar o entendimento.

```python
    def enquadrar(self, bits: list[int]) -> list[int]:
        count = 0
        stuffed = []
        for b in bits:
            stuffed.append(b)
            if b == 1:
                count += 1
                if count == 5:
                    stuffed.append(0)  # bit stuffing
                    count = 0
            else:
                count = 0
        return self.FLAG + stuffed + self.FLAG

    def desenquadrar(self, quadro: list[int]) -> list[int]:
        dados = quadro[8:-8]  # remove FLAGs
        count = 0
        result = []
        i = 0
        while i < len(dados):
            b = dados[i]
            result.append(b)
            if b == 1:
                count += 1
                if count == 5:
                    i += 1  # ignora o bit inserido (0)
                    count = 0
            else:
                count = 0
            i += 1
        return result
```

Figura 16. Parte do script utilizada para implementação do protocolo de enquadramento de dados utilizando enquadramento com FLAGS e inserção de bits.

O trecho de código acima implementa o enquadramento por inserção de bits (bit stuffing). Ele insere um bit 0 após cada sequência de cinco 1s consecutivos nos dados, para evitar que sejam confundidos com as FLAGs de início/fim do quadro. No desenquadramento, esses 0s extras são removidos, restaurando a mensagem original.

B. Protocolo de Detecção de Erros - Desenvolvido por: Henrique Givisiez

1) Bit de paridade

É a estratégia mais simples que permite a detecção de erros individuais, consiste em acrescentar aos dados um único bit de paridade. Nesse sentido, o bit de paridade é escolhido de forma que o número número de bits 1 seja par ou impar. Do ponto de vista do receptor, este recebe o trem de bits e retira o bit de paridade, após isso checa a paridade, caso seja igual não houve erro (ou não foi possível detectar) se for diferente teve erro. Vale ressaltar que detecta apenas um bit de erro. A seguir, seguirá um trecho do código a fim de detalhar a implementação.

2) CRC (polinômio CRC-32, IEEE 802)

Tanto o transmissor quanto o receptor devem utilizar o mesmo polinômio gerador. O transmissor usa este polinômio para calcular um código de redundância cíclica (CRC) que é anexado aos dados. O receptor realiza o mesmo cálculo e compara o resultado com o CRC recebido para detectar erros na transmissão. O CRC-32 (especificado pelo IEEE 802) é uma versão amplamente utilizada deste método. A seguir, segue um trecho do código implementado, a fim de facilitar o entendimento acerca da implementação.

C. Protocolo de Correção de Erros (Hamming) - Desenvolvido por: Gabriel Castro

Neste trabalho, foi feita a implementação simplificado do código de Hamming, este trata de blocos de 7 bits de dados, aos quais são adicionador 4 bits de paridade para formar um código de 11 bits. Este esquema permite a detecção e

```python
    def transmitir(self, mensagem: list[int]) -> list[int]:
        """
        Metodo responsavel por transmitir uma mensagem, adicionando o bit de paridade par.

        Args:
        mensagem (list[int]): Lista de bits (0 ou 1) representando a mensagem original.

        Returns:
        list[int]: Lista de bits com o bit de paridade adicionado ao final.
        """
        # Conta quantos bits '1' existem na mensagem
        conta_uns = mensagem.count(1)

        # Calcula o bit de paridade: se eh quantidade par de '1's, paridade eh 0;
        # caso contrario, adiciona 1 para tornar par.
        paridade = 0 if conta_uns % 2 == 0 else 1

        # Adiciona o bit de paridade ao final da mensagem
        mensagem.append(paridade)

        # Retorna a mensagem com o bit de paridade
        return mensagem

    def verificar(self, mensagem: list[int]) -> bool:
        """
        Metodo responsavel por verificar se a paridade da mensagem esta correta.

        Args:
        mensagem (list[int]): Lista de bits com o bit de paridade ja incluido.

        Returns:
        bool: True se a paridade estiver correta (numero total de '1's eh par),
              False caso contrario.
        """
        # Conta o número total de bits '1' (incluindo o bit de paridade)
        total_uns = mensagem.count(1)

        # Retorna True se a quantidade for par (paridade válida), False se for impar
        return total_uns % 2 == 0
```

Figura 17. Parte do script utilizada para implementação do protocolo de detecção de erros utilizando Bit de paridade.

```python
    def transmitir(self, mensagem: list[int]) -> list[int]:
        """
        Gera o codigo CRC para a mensagem e retorna a mensagem original com os bits de CRC anexados.

        Args:
        mensagem (list[int]): Lista de bits da mensagem original.

        Returns:
        list[int]: Mensagem original seguida do codigo CRC (32 bits).
        """
        # Converte a lista de bits para inteiro
        dados = self._lista_para_inteiro(mensagem)

        # Desloca os bits a esquerda para reservar espaço para o CRC (grau do polinômio)
        dados <<= self.GRAU

        # Calcula o CRC usando divisao polinomial
        for i in reversed(range(len(mensagem))):
            if (dados >> (i + self.GRAU)) & 1:
                dados ^= self.POLINOMIO << i

        # O CRC esta nos bits inferiores apos o XOR
        crc = dados & ((1 << self.GRAU) - 1)

        # Concatena os bits originais com o CRC
        return mensagem + self._inteiro_para_lista(crc, self.GRAU)

    def verificar(self, mensagem: list[int]) -> bool:
        """
        Verifica se o codigo CRC da mensagem esta correto.

        Args:
        mensagem (list[int]): Lista de bits contendo a mensagem original + CRC (ultimos 32 bits).

        Returns:
        bool: True se o CRC for valido (sem erro), False se houver erro.
        """
        # Converte a lista de bits para inteiro
        dados = self._lista_para_inteiro(mensagem)

        # Executa a divisao polinomial para verificar o CRC
        for i in reversed(range(len(mensagem) - self.GRAU)):
            if (dados >> (i + self.GRAU)) & 1:
                dados ^= self.POLINOMIO << i

        # Se o resultado for 0, o CRC eh valido
        return (dados & ((1 << self.GRAU) - 1)) == 0
```

Figura 18. Parte do script utilizada para implementação do protocolo de detecção de erros utilizando CRC.

correção de erros de um único bit que possam ocorrer durante a transmissão. No processo de codificação, os bits de paridade são calculados e inseridos em posições específicas com base nos bits de dados. Na recepção, os bits de paridade são recalculados para identificar a posição de qualquer erro, que pode então ser corrigido, garantindo assim a integridade da informação transmitida. A seguir, está o trecho do código principal que é responsável por determinar a posição do erro, o cerne da capacidade de correção do Hamming:

```python
        # Comparo os bits de paridade calculados com o recebido
        # p/ o bit 1
        if p1_recebido != p1_recalculado:
            s1 = 1
        else:
            s1 = 0
        # p/ o bit 2
        if p2_recebido != p2_recalculado:
            s2 = 1
        else:
            s2 = 0
        # p/ o bit 4
        if p4_recebido != p4_recalculado:
            s4 = 1
        else:
            s4 = 0
        # p/ o bit 8
        if p8_recebido != p8_recalculado:
            s8 = 1
        else:
            s8 = 0

        # Calculo a posição que tem erro (calculando o binário baseado nos grupos que tem erro)
        posicao_com_erro = s1 * 1 + s2 * 2 + s4 * 4 + s8 * 8
        return posicao_com_erro
```

Figura 19. Parte do script utilizada para implementação do protocolo de correção de erros utilizando Hamming simplificado.

Note que a posição do erro é determinada através da recalculação dos bits de paridade no receptor. Comparando esses valores recalculados com os bits de paridade recebidos, geramos bits de síndrome. Nesse sentido, a combinação binária desses bits de síndrome (por exemplo, $s1, s2, s4, s8$) forma um número que corresponde diretamente ao índice do bit com erro. Se todas as síndromes forem zero, não há erro detectado.

IV. Interface Gráfica - Desenvolvida por: Henrique Givisiez e André Modesto

Para a construção da interface gráfica do projeto, optou-se pela utilização da biblioteca GTK (GIMP Toolkit). Essa interface visa proporcionar uma visualização intuitiva e interativa das simulações. Além disso, ela permite a configuração de parâmetros da camada de enlace e da camada física como o méotodo de enquadramento, o tipo de modulação e até mesmo o tamanho máximo da carga útil dos quadros.

O transmissor e o receptor foram implementados em janelas distintas, com campos para entrada e saída de dados, botões para iniciar a simulação e elementos visuais para exibir os sinais modulados em gráficos gerados com a biblioteca matplotlib. A arquitetura modular da interface facilita a integração com o simulador principal, garantindo que cada camada seja acionada de acordo com as escolhas do usuário. A separação entre transmissor e receptor também simula de forma realista o comportamento em um ambiente de rede. No que diz respeito a comunicação entre o transmissor e o receptor, ela foi implementada via sockets TCP, simulando um meio de transmissão confiável. Isso permite que os dados modulados

sejam enviados de forma realista entre os dois módulos da aplicação.

Abaixo, é possível observar as janelas do transmissor e do receptor da interface desenvolvida.

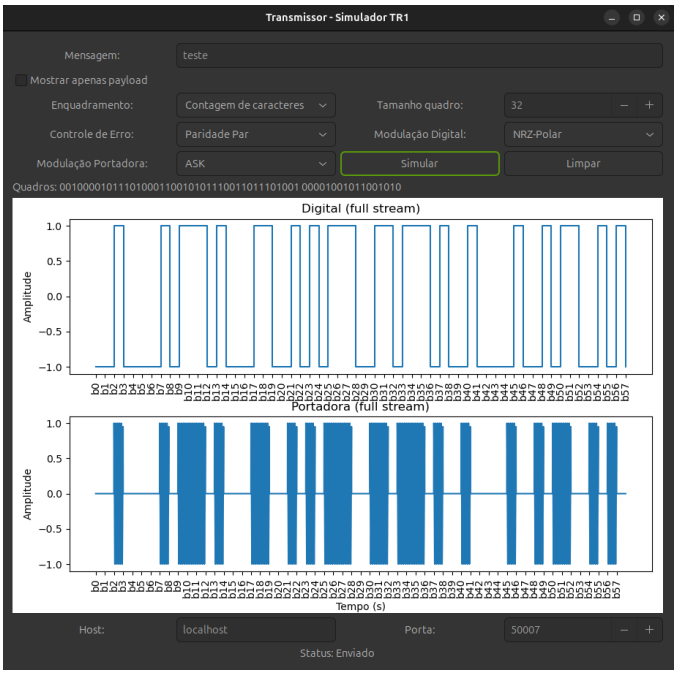

Figura 20. Interface transmissor.

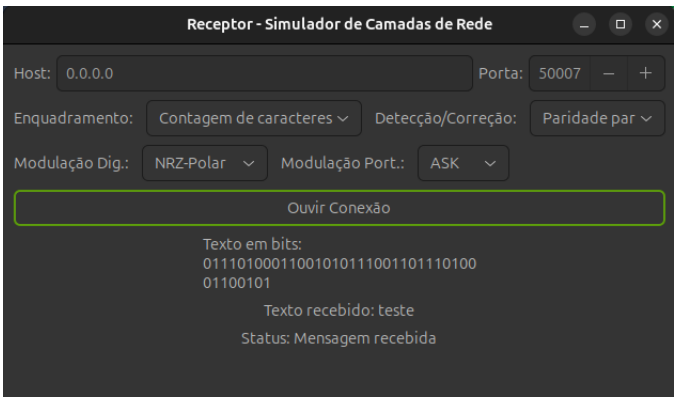

Figura 21. Interface receptor.

V. Conclusão

Este projeto demonstrou a simulação do funcionamento das camadas de enlace e física em uma comunicação digital, abrangendo desde o enquadramento de dados até as diferentes técnicas de modulação. A maior complexidade, e ao mesmo tempo o ponto de maior aprendizado, residiu na integralização da interface gráfica, que conectou de forma coesa as funcionalidades do transmissor e do receptor. Foi por meio dessa interface que se tornou possível visualizar e compreender de forma intuitiva as transformações dos dados em cada etapa do processo de comunicação, solidificando o entendimento dos conceitos teóricos e práticos abordados. A capacidade de interagir com o sistema e observar os sinais em tempo real foi crucial para a análise e validação dos protocolos implementados.

Referências

[1] Python Software Foundation. *Abstract Base Classes (abc)*. Disponível em: https://docs-python-org.translate.goog/3/library/abc.html?_x_tr_sl=en&_x_tr_tl=pt&_x_tr_hl=pt&_x_tr_pto=tc. Acesso em: 05/07/2025.

[2] The GTK+ Project. *Python GTK+ 3 Tutorial*. Disponível em: https://python-gtk-3-tutorial.readthedocs.io/en/latest/. Acesso em: 05/07/2025.

[3] PEREIRA, Geraldo. *Geraldo Pereira* (Canal no YouTube). Disponível em: https://www.youtube.com/watch?v=TjSayD845KI. Acesso em: 05/07/2025.

[4] MAROTTA, Marcelo. *Marcelo Marotta* (Canal no YouTube). Disponível em: https://www.youtube.com/watch?v=502xv4aOep4. Acesso em: 05/07/2025.