Universidade de Brasília

INSTITUTO DE EXATAS - IE

DEPARTAMENTO DE CIÊNCIAS DA COMPUTAÇÃO - CIC

CIC0235 - TELEINFORMÁTICA E REDES 1 1/2025

NOME DO SIMULADOR: “Physical-DataLink-Sim"

TRABALHO FINAL – GRUPO G7

Tema: Simulador das Camadas Física e de Enlace em Redes de Computadores	**Rodrigo Fonseca Torreao (211066196)**
	Arthur Delpino Barbabella (221002094)
	João Victor Cavallin Pereira (212008894)

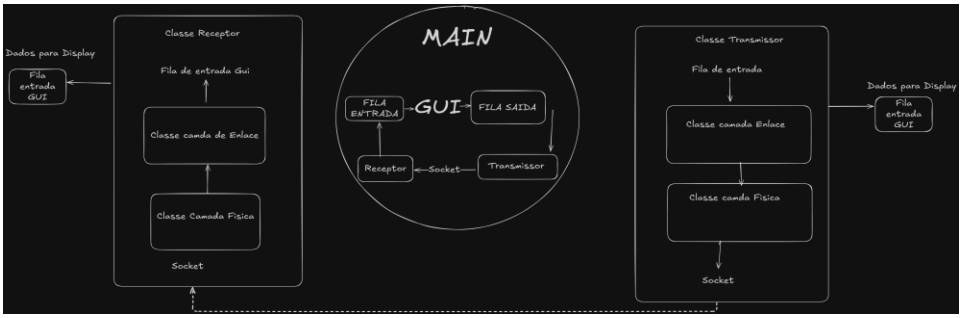
1 INTRODUÇÃO

Em redes de computadores, a comunicação confiável entre dispositivos depende fundamentalmente do funcionamento coordenado da camada física (responsável pela transmissão de sinais no meio físico) e da camada de enlace (encarregada do controle de fluxo, enquadramento e integridade dos dados). Este trabalho visa implementar um simulador didático que integra protocolos essenciais dessas camadas, permitindo a análise prática de:

- Modulação digital (NRZ-Polar, Manchester e Bipolar) e modulação por portadora (ASK, FSK, 8-QAM) na camada física;
- Enquadramento de dados (contagem de caracteres, inserção de bytes e bits com FLAGS);
- Detecção de erros (paridade par e CRC-32);
- Detecção e correção de erros (código de Hamming).

1.1 Visão Geral do Simulador:

O simulador desenvolvido segue um fluxo unidirecional (transmissor e receptor), conforme ilustrado na Figura 1 do trabalho:



1.1.1 Simulador:

(A) DESCRIÇÃO GERAL

a. Arquivo *main* do programa. Responsável por iniciar todas as *threads* e garantir o fluxo de dados correto. Essas *threads* permitem que o transmissor e o receptor funcionem simultaneamente, imitando o comportamento de processos paralelos em uma rede real.

b. Cada módulo é executado dentro de uma *thread* separada:
   - A *thread* do transmissor aguarda os dados da interface gráficas, aplica o enquadramento e EDC, e envia um dicionário serializado em *bytes* para o receptor, contendo os parâmetros de comunicação, sinal digital e analógico.
   - A thread do receptor aguarda o quadro transmitido, faz a demodulação digital, realiza a verificação do EDC, desenquadra os quadros e envia o resultado para a interface gráfica de recepção.
   - Enquanto a interface gráfica está rodando na *thread* principal, as *threads* do transmissor e receptor rodam em paralelo, processando dados e se comunicando com a *GUI* pelas filas (*Queues*).

1.1.2 Interface GUI:

(B) DESCRIÇÃO GERAL

a. Desenvolvido utilizando a biblioteca GTK. Recebe dados estruturados (bits, strings e figuras) através de uma fila de entrada, e os direciona para o campo correto de apresentação. Além disso, possui uma fila de saída, onde são postados os dados a serem simulados (mensagem a ser enviada, tipo de modulação, nível de ruído etc.).

b. Cada janela possui abas separadas para exibir informações de cada camada:
   - **Camada de Aplicação**: mostra os dados originais e decodificados;
   - **Camada de Enlace**: mostra os quadros gerados e processados;
   - **Camada Física**: mostra os gráficos resultantes das modulações.

1.1.3 Transmissor:

(C) DESCRIÇÃO GERAL

a. Recebe os dados da mensagem via a fila de saída da interface gráfica. É responsável por aplicar os protocolos da camada enlace e física. Em seguida envia os dados ao receptor via *socket* e publica as imagens do sinal gerado na fila de entrada da interface gráfica.

b. Ao perceber alguma mensagem da *in_queue*, o transmissor executa as seguintes etapas:
   - **Camada de Aplicação:** converte a mensagem de texto em *bytes utf-8* e exibe o texto original e sua representação em bits;
   - **Camada de Enlace:** aplica o tipo de enquadramento selecionado e, em seguida, insere o EDC escolhido na GUI (CRC, Hamming ou bit de paridade). Todo o processo é exibido na interface gráfica;
   - **Camada Física:** aplica o tipo de codificação de banda base selecionado (NRZ-Polar, Bipolar ou Manchester) e de modulação por portadora (ASK, FSK ou 8-QAM) no quadro processado. Logo após, exibe os sinais obtidos em gráficos na interface;

c. Ao final da execução do método principal *start()*, o transmissor envia o quadro processado e os sinais gerados para o receptor via *socket*, usando o protocolo TCP na porta local 711. Os dados são organizados em um dicionário *msg_dict*, que é serializado em *bytes* usando o módulo *pickle* e enviado para a conexão.

1.1.4 Receptor

(A) DESCRIÇÃO GERAL

a. O arquivo receptor.py atua como ponto final da comunicação em um sistema baseado em TCP/IP, sendo responsável por receber, interpretar e exibir os sinais transmitidos pelo transmissor;
b. Após o estabelecimento da conexão, os dados são recebidos em blocos de bytes, desserializados utilizando a biblioteca *pickle* e transformados em um dicionário contendo os sinais (analógico e digital) e os parâmetros da transmissão;
c. O processamento ocorre em três etapas:
   - Exibição do sinal analógico recebido com **aplicação de ruído**;
   - O sinal digital é demodulado para banda base, esse processo utiliza o tipo de demodulação aplicado na transmissão. (Ex: NRZ, Manchester);
   - O quadro demodulado é processado para desenquadramento, também é realizado a verificação de integridade, fazendo uso de códigos de detecção de erros (EDC), tais como CRC, Hamming ou bit de paridade.
d. Durante todo o processo, a classe atualiza a GUI com visualizações gráficas dos sinais, quadros intermediários e mensagens de status

1.1.5 Aplicação de Ruído no Receptor

(B) DESCRIÇÃO GERAL

a. No programa, o ruído é aplicado sobre o sinal analógico e digital recebidos pelo receptor, de acordo com o Nível de Ruído fornecido como entrada na GUI e repassada à *in_queue* do transmissor.

b. A dinâmica da aplicação do ruído consiste no uso da função *np.random.normal(0, sigma, len(sinal_analógico))*, do módulo *numpy*, que gera um vetor *numpy.array* com valores de ruído contidos em uma Distribuição Gaussiana de média zero e desvio padrão sigma σ , para cada amostra do sinal analógico. Ao somar o vetor *numpy* do ruído com o do sinal analógico, aplica-se o ruído em cada um de seus pontos. No entanto, não é possível repetir esse processo sobre o sinal digital, que tem um número de elementos na ordem $1:100$ do analógico, afinal os valores de ruído seriam diferentes, porém dentro da mesma curva.

c. Para garantir que os níveis de tensão do sinal digital sofressem o mesmo ruído, foi tirada a média de ruído de cada 100 elementos do vetor de ruído, associados a um único bit do quadro modulado e, por consequência, a um único nível de tensão também. A partir disso, foi constituído um vetor de ruído equivalente para o sinal digital, com $\frac{1}{100}$ do número de elementos do vetor de ruído do sinal analógico, e aplicado em cima dos níveis de tensão.

2 CAMADA FÍSICA

A camada física é responsável pela transmissão de dados através do meio físico, estabelecendo como os bits são convertidos em sinais elétricos, ópticos ou de rádio. É nessa camada que ocorre a modulação digital e analógica, de forma a tornar os bits aptos a serem enviados no meio. No projeto, foram implementadas as técnicas de modulação digital NRZ-Polar (Non-Return to Zero Polar), Bipolar e Manchester e as técnicas de modulação por portadora 8-QAM, ASK (Amplitude Shift Keying) e FSK (Frequency Shift Keying).

2.1 MODULAÇÃO DIGITAL

Este trabalho implementou as codificações em banda base NRZ-Polar, Manchester e Bipolar, descritas nos subtópicos a seguir.

2.1.1 Non-return to Zero Polar (NRZ-POLAR)

(D) DESCRIÇÃO GERAL

a. Codifica dados utilizando a codificação banda base NRZ-Polar (Non-Return to Zero Polar), onde cada bit da mensagem é convertido em um nível de tensão:

$$Bit\ 0 \rightarrow -1V$$
$$Bit\ 1 \rightarrow +1V$$

(E) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE CODIFICAÇÃO:

codificar_nrz_polar$(dado\colon bytes) \rightarrow List[int]$

i. **Entrada:** *(tipo: bytes)* Um quadro da camada de enlace.
Ex: b"A" $\rightarrow$ 01000001

ii. **Saída:** *(tipo: List[int])* Lista com valores de tensão correspondentes aos *bits* codificados.
Ex: $[-1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ -1]$

iii. Passo a passo:

1. Converte bytes em *string* de *bits*;
2. Faz mapeamento da codificação;
3. Retorna lista com sinal digital.

b. FUNÇÃO DE DECODIFICAÇÃO:

decodificar_nrz_polar(sinal_digital:list) $\rightarrow bytes$

i. **Entrada:** *(tipo: List[int])* Lista com valores de tensão correspondentes aos bits codificados.
Ex: $[-1,\ 1,\ -1,\ -1,\ -1,\ -1,\ 1,\ -1]$

ii. **Saída:** *(tipo: bytes)* Um quadro da camada de enlace.
Ex: b"A" $\rightarrow$ 01000001

iii. Passo a passo:

1. Aproxima a tensão após aplicação do ruído no receptor;
2. Desfaz o mapeamento da NRZ-Polar;
3. Converte quadro de *string* de *bits* para inteiro na base 2;
4. Obtém quadro original, em *bytes*;
5. Retorna quadro, em *bytes*.

2.1.2 Manchester

(F) DESCRIÇÃO GERAL

a. Codifica dados utilizando a codificação Manchester. Nela, cada *bit* é expandido em dois e é feito *XOR* com o sinal de clock alternado (0, 1). Utiliza-se o sinal de *clock* para definir qual a tensão associada ao *bit* a ser transmitido, da seguinte forma:

- Quando o *bit* é 0, o sinal digital equivalente é uma borda de subida [0V, 1V]
- Quando o *bit* é 1, o sinal digital equivalente é uma borda de descida [1V, 0V]

(G) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE CODIFICAÇÃO:

$$codificar\_manchester(dado: bytes) \rightarrow List[int]$$

i. **Entrada:** *(tipo: bytes)* Um quadro.
Ex: b"A" $\rightarrow$ 01000001

ii. Saída: *(tipo: List[int])* Lista com valores de tensão correspondentes aos *bits* codificados.
Ex: $[0,1,1,0,0,1,0,1,0,1,0,1,0,1,1,0]$

iii. Passo a passo:

1. Converte *bytes* em *string* de *bits*;
2. Gera o sinal de *clock* ("01") para cada bit do quadro;
3. Para cada *bit*, após conversão em inteiro, realiza XOR com o sinal de *clock* (01);
4. Armazena 2 *bits* resultantes para cada *bit* na lista de resultado;
5. Retorna lista com sinal digital.

b. FUNÇÃO DE DECODIFICAÇÃO

$$decodificar\_manchester(sinal\_digital:list) \rightarrow bytes$$

i. Entrada: *(tipo: List[int])* Lista com valores de tensão correspondentes aos bits codificados.
Ex: $[0,1,1,0,0,1,0,1,0,1,0,1,0,1,1,0]$

ii. Saída: *(tipo: bytes)* Um quadro.
Ex: b"A" $\rightarrow$ 01000001

iii. Passo a passo:

1. Para cada par de bits:
   - Se o par for [1, 0], recupera o bit 1;
   - Se o par for [0, 1], recupera o bit 0;
2. Agrupa os bits em blocos de 8 para formar os bytes originais do quadro;
3. Retorna os bytes reconstruídos a partir do sinal decodificado.

2.1.3 Bipolar

(H) DESCRIÇÃO GERAL

a. A codificação bipolar, também chamada de AMI (Alternate Mark Inversion), é modulada de acordo com os 1's e apresenta valor 0V para os bits 0's:

$$Bit\ 0 \rightarrow 0V$$

$$Bit\ 1 \rightarrow \pm 1V$$

(I) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE CODIFICAÇÃO

codificar_bipolar(quadro: bytes) -> list

i. **Entrada:** *(tipo: bytes)* Um quadro da camada de Enlace
Ex: b"N" $\rightarrow$ 01001110

ii. **Saida:** *(tipo: list[int])* Lista com valores de tensão correspondente a codificação
Ex: $[0,\ 1,\ 0,\ 0,\ -1,\ 1,\ -1,\ 0]$

iii. Passo a passo:

1. Converte os bytes em string de bits;
2. Inicializa a lista e a polaridade do sinal;
3. A codificação é feita por meio de um laço que percorre a string de bits;
4. Quando bit é 1, adiciona a polaridade atual na lista e em seguida inverte a polaridade (Ex: $1 \rightarrow -1$), quando bit é 0, apenas adiciona 0 a lista.

b. FUNÇÃO DE DECODIFICAÇÃO

decodificar_bipolar(sinal: list) -> bytes

i. **Entrada:** *(tipo: List[int])* Lista com os valores de tensão correspondentes aos bits codificados
Ex: $[0,\ 1,\ 0,\ 0,\ -1,\ 1,\ -1,\ 0]$

ii. **Saida:** *(tipo: bytes)* Um quadro da camada de enlace
Ex: b"N" $\rightarrow$ 01001110

iii. **Passo a passo:**

1. Aproxima a tensão após aplicação de ruído;
2. Constrói a sequência de bits;
3. Agrupa em blocos;
4. Converte os blocos para bytes.

2.2 MODULAÇÃO POR PORTADORA

Este trabalho implementa três técnicas de modulação por portadora em cima dos quadros: *Amplitude Shift Keying (ASK)*, *Frequency Shift Keying (FSK)* e *8-Quadrature Amplitude Modulation (8-QAM),* conforme descrito nos subtópicos a seguir.

2.2.1 Amplitude Shift Keying (ASK)

(A) DESCRIÇÃO GERAL

a. Essa técnica de modulação digital consiste em alterar a amplitude da onda portadora de acordo com os dados de entrada. A presença de sinal é representada por um aumento na amplitude da onda portadora, enquanto a ausência de sinal é representada pela diminuição da amplitude do sinal.

(B) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE CODIFICAÇÃO

modular_ask($quadro$: $bytes$, $amostras\_por\_bits$: $int = 100$, fs: $int = 800$) → $list$

i. **Entrada:** *(tipo: bytes)* Um quadro da camada de enlace.
Ex: b"A" → 01000001

ii. **Saida:** *(tipo: List[float])* Lista com os valores do sinal modulado, onde bits 1 geram uma senoide e bits 0 geram amplitude zero.
Ex: [0.0, 0.0628. 0.1253...]

iii. **Passo a passo:**

1. Converte *bytes* em *string* de *bits*;
2. Calcula a frequência portadora: $freq = \frac{fs}{amostras\_por\_bits}$
3. Define o intervalo de amostragem: $dt = \frac{1}{fs}$
4. Inicializa $tempo\_acumulado$ e o vetor de saída
5. Percorre cada bit e gera sinal correspondente, para bit 1 gera uma senoide de amplitude 1, para bit 0 gera um sinal de amplitude zero;
6. Atualiza $tempo\_acumulado \mathrel{+}= dt$ para garantir a continuidade e o alinhamento da senoide.

2.2.2 Frequency Shift Keying (FSK)

(C) DESCRIÇÃO GERAL

a. Essa técnica de modulação analógica consiste em modificar a frequência da portadora $f(x) = A \times \sin(2\pi f)$ de acordo com o *bit* a ser transmitido, da seguinte forma:

- Caso o *bit* seja 0, a frequência da portadora no **tempo de bit** é de 2 Hz.
- Caso o *bit* seja 1, a frequência da portadora no **tempo de bit** é de 5 Hz.

(D) IMPLEMENTAÇÃO EM PYTHON

a. **FUNÇÃO DE CODIFICAÇÃO**

$$\text{modular\_fsk}(quadro: bytes, f0 = 2, f1 = 5, amostras = 100, fs = 800) \rightarrow List[float]$$

iv. **Entrada:** *(tipo: bytes)* Um quadro da camada de enlace.
Ex: b"A" $\rightarrow$ 01000001

v. **Saída:** *(tipo: List[float])* Lista com os pontos/amostras que representam cada *bit* (100 pontos por *bit*).
Ex: [0, 0, 0.0627, 0.125...]

vi. Passo a passo:

1. Converte *bytes* em *string* de *bits*;
2. Define tempo de amostra $dt = \frac{1}{fs}$, em que fs é a frequência de amostragem (*default*=800), que define o detalhamento do gráfico;
3. Para cada *bit*, serão gerados 100 pontos (*float*) de acordo com a sua frequência, dados pela operação $f(x) = A \times \sin(2\pi f \times dt)$. A fase $2\pi f \times dt$ acumula e leva ao próximo ponto da onda;
4. Retorna uma lista com os pontos da senoide que representa os *bits* do quadro.

2.2.3 8-Quadrature Amplitude Modulation (8-QAM)

(E) DESCRIÇÃO GERAL

a. Esse método de modulação por portadora consiste na transmissão de símbolos elétricos que representam, de uma vez, 3 *bits*. Os 3 *bits* são mapeados em dois componentes:
   - **I *(In-Phase)*:** Parte em fase do sinal, relacionada ao cosseno.
   - **Q *(Quadrature)*:** Parte em quadratura, relacionada ao seno.

(F) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE CODIFICAÇÃO

$$\textit{modular\_8qam} = \left(quadro: bytes,\ amostras_{por_{simbolo}} \coloneqq 100,\ fs = 800\right) \rightarrow List[float]$$

i. **Entrada:** *(tipo: bytes)* Um quadro.
Ex: *b"a"* $\rightarrow$ 01000001

ii. Saída: (*tipo: List(float)*) Lista com os pontos/amostras por símbolo elétrico (100 pontos por símbolo ao invés de por *bit*). Ex: [-1, -0.874, -0.719...]

iii. Passo a passo:

1. Converte bytes em *string* de *bits*;
2. Monta o *rangeMap*, em que cada combinação de 3 *bits* é associada um par (I, Q). Esses pares representam pontos no plano IQ da modulação.
3. Força o número total de *bits* a ser múltiplo de 3, para que todos os símbolos elétricos estejam completos. Usa *padding* inserindo zeros ao final, caso não seja múltiplo de 3;
4. Calcula o tempo de amostra $dt = \frac{1}{fs}$e a frequência para gerar 1 ciclo completo por símbolo $freq = \frac{fs}{800}$;
5. Faz o mapeamento do símbolo para o par (I, Q), de acordo com o *rangeMap*;
6. Obtém as 100 amostras para cada símbolo (3 bits), considerando que para cada amostra $s = I \times \cos(fase) + Q \times \sin(fase)$, na qual a fase segue sendo dada por $2\pi ft$;
7. Retorna uma lista com os pontos da senoide que representa os *bits* do quadro.

3 CAMADA DE ENLACE

A camada de enlace é responsável pela organização dos dados recebidos, garantindo a detecção e correção de erros, o controle de fluxo e o enquadramento dos dados. Dessa forma, os dados transmitidos da camada física são interpretados corretamente.

No projeto, foram implementados os protocolos de **enquadramento contagem de caracteres**, **enquadramento com *flags* e inserção de *bytes*** e o **enquadramento com flags de inserção de *bits***. Para a detecção de erros, foram implementadas as técnicas de ***bit* de paridade par** e **CRC** (polinômio CRC-32, IEEE 802). A correção de erros implementada foi **Hamming (7,4)**.

3.1 PROTOCOLOS DE ENQUADRAMENTO

Os protocolos de enquadramento definem com os dados são agrupados em quadros. Eles garantem que o início e o fim de cada quadro sejam corretamente identificados, mesmo em fluxos contínuos de bits.

3.1.1 Contagem de caracteres

(A) DESCRIÇÃO GERAL

a) A contagem de caracteres adiciona ao início de cada quadro um *byte* que informa o tamanho total do quadro, incluindo o *próprio byte.*

(B) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE ENQUADRAMENTO

enquadrar_contagem_caracteres(dado: bytes) → *bytes*

i. **Entrada:** *(tipo: bytes)* Dados da camada de aplicação a serem enquadrados.
Ex: *b"teste"*

ii. **Saída:** *(tipo: bytes)* Um quadro.
Ex: *b"\x06teste"*

iii. Passo a passo

1. Calcula o tamanho total do quadro adicionado o *byte* de cabeçalho;
2. Converte o número inteiro em 1 *byte*, no padrão *big-endian*;
3. Concatena o *byte* do tamanho do quadro com o quadro.

b. FUNÇÃO DE DESENQUADRAMENTO

denquadrar_contagem_caracteres(quadro: bytes) -> byte

i. **Entrada:** *(tipo: bytes)* Quadro com o byte de tamanho seguido do *payload;*
Ex: *b"\x06teste"*

ii. **Saída:** *(tipo: bytes)* Apenas o *payload*;
Ex: *b"teste"*

iii. Passo a passo

1. Remove o byte que indica o tamanho do quadro e retorna o dado da camada de aplicação.

3.1.2 Enquadramento com FLAGS e inserção de bytes

(C) DESCRIÇÃO GERAL

a) O enquadramento por inserção de *bytes* delimita o início e o fim de uma mensagem utilizando a flag *b'\x7E',* para evitar confusão em casos que a própria *flag* aparece dentro do dado, utiliza-se o caractere de escape *'b\x7D'.* Dessa forma, sempre que o a flag ou o escape aparecem nos dados, ele será precedido pelo próprio caractere de escape.

(D) IMPLEMENTAÇÃO EM PYTHON

c. FUNÇÃO DE ENQUADRAMENTO

enquadrar_flag_insercao_byte(dado:bytes, flag=b'\x7E', esc=b'\x7D') -> bytes:

i. **Entrada:**
   a. *dado (tipo: bytes):* sequência de dados da camada de aplicação que será enquadrada;
   b. *flag (b'\x7E'):* delimitador início/fim do quadro;
   c. *esc (b'\x7D'):* caractere de escape.

   Ex: *b"teste"*

ii. **Saída:** *(tipo: bytes)* Quadro enquadrado.
   Ex: *b"\x7Eteste\x7E"*

iii. Passo a passo

1. Verifica se a flag aparece no dado;
2. Localiza os caracteres de escape e, se existir, adiciona imediatamente antes outro caractere de escape;
3. Localiza as flags no dado e, se existir, adiciona o caractere de escape imediatamente antes;
4. Retorno o dado com uma flag e outra depois. $FLAG + DADO + FLAG$

d. FUNÇÃO DE DESENQUADRAMENTO

desenquadrar_flag_insercao_byte(quadro: bytes, flag=b'\x7E', esc='\x7D') -> bytes:

i. **Entrada:**
   a. *quadro (tipo: bytes):* Quadro com flag e caractere de escape;
   b. *flag (b'\x7E'):* delimitador início/fim do quadro;
   c. *esc (b'\x7D'):* caractere de escape.

   Ex: *b"\x7Eteste\x7E"*

ii. **Saída:** *(tipo: bytes)* Dados originais.
   Ex: Ex: *b"teste"*

iii. Passo a passo

1. Decodifica o dado para UTF-8;
2. Verifica se o tamanho do quadro é maior que $2 \times FLAG$ e se o quadro possui flag no início e no fim, caso qualquer um desses casos for falso, o código lança um erro;
3. Remove a flag inicial e final do quadro;

4. Inicia um laço que percorre cada caractere:
    - Caso encontre algum caractere de escape, ignora, seleciona o próximo caractere e adiciona ao resultado (caso não encontre um próximo caractere, lança um erro de espace incompleto);
    - Caso não encontre um caractere de escape, adiciona o próprio caractere que está sendo percorrido.

3.1.3 Enquadramento com FLAGS Inserção de bits

(E) DESCRIÇÃO GERAL

a) O enquadramento por inserção de bits delimita o início e o fim de uma mensagem utilizando a flag *b "\x7E"*. Para evitar que essa flag apareça dentro dos dados, insere-se um 0 após a sequência de 5 bits 1 consecutivos.

(F) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE ENQUADRAMENTO

enquadrar_flag_insercao_bit(dado:bytes) -> bytes

i. **Entrada:** *dado (tipo: bytes):* Conteúdo da carga util.
ii. **Saída:** (tipo: bytes) Quadro enquadrado com flag e bit stuffing
iii. **Passo a passo**
1. Converte os dados em string de bits;
2. Percorre cada bit da mensagem original, para cada bit percorrido adiciona à lista que armazena o enquadramento, porém conta os bits 1 consecutivos e, em caso de 5 bits consecutivos, insere um bit 0;
3. Preenche o quadro com bits 0 para garantir um tamanho de quadro múltiplo de 8;
4. Converte essa string para bytes
5. Retorna: $FLAG + BYTES\_PREENCHIDOS + FLAG$

b. FUNÇÃO DE DESENQUADRAMENTO

desenquadrar_flag_insercao_bit(quadro:bytes) -> bytes

i. **Entrada:** quadro *(tipo: bytes):* Quadro com flags e bit stuffing
ii. **Saída:** (tipo: bytes) Dados originais
iii. **Passo a passo**

1. Verifica se inicia e termina com a flag delimitadora, caso sim remova as flags caso não, lança um erro de delimitação ausente;
2. Converte o quadro para uma string de bits;
3. Percorre a string bit a bit e insere ao resultado enquanto conta os bits 1 consecutivos, quando atingir 5 bits consecutivos, ignora o próximo bit (0 inserido por stuffing)
4. Remove os bits adicionados para alinhamento;
5. Converte os bits para bytes novamente;
6. Retorna o dado original.

3.2 PROTOCOLOS DE DETECÇÃO DE ERROS

Os protocolos de detecção de erros são um conjunto de técnicas usadas para identificar alterações nos dados durante a transmissão. Dessa forma permitindo ao receptor reconhecer que o quadro chegou com erro, mesmo que não consiga corrigi-lo.

3.2.1 Bit de paridade par

(G) DESCRIÇÃO GERAL

a. Técnica simples de *EDC* (*Error detection code*), que força o número de *bits* 1 do quadro a ser par antes da transmissão, da seguinte forma:
   - Caso o número de *bits* 1 seja ímpar: acrescenta o *bit* 1 ao final do quadro.
   - Caso o número de *bits* seja par: acrescenta o *bit* 0 ao final do quadro.

b. O processo de verificação, na ponta do receptor, consiste simplesmente em verificar se o número de *bits* 1 recebidos é ímpar.

c. O projeto, na camada de enlace, trabalha exclusivamente com os quadros em *bytes*. Por isso, o *bit* de paridade par é acrescentado ao quadro dentro de um novo *byte*, no *bit* menos significativo.

(H) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE EDC

$$bit\_de\_paridade\_par(quadro\colon bytes) \rightarrow bytes$$

i. **Entrada:** Um quadro, em *bytes*.
Ex: b"s" → 01110011

ii. **Saída:** Quadro com o *bit* de paridade
Ex: 01110011 00000001

iii. Passo a passo:

1. Converte *bytes* em *string* de *bits*;

2. Adiciona o *byte* com o LSB 1 se o número de *bits* 1 for ímpar, ou um *byte* com o LSB 0 caso seja par;
3. Retorna o quadro com um novo *byte*, ao final, que contém o *bit* de paridade par no LSB.

b. FUNÇÃO DE VERIFICAÇÃO

$$verifica\_crc(quadro\colon bytes,\ tamanho\colon int = 8,\ polinomio\colon int = 0x07) \rightarrow bytes$$

i. **Entrada:** Quadro com o bit de paridade.
Ex: b"s\x01"

ii. **Saída:** Quadro sem o bit de paridade.
Ex: b"s"

iii. Passo a passo:

1. Converte bytes em string de bits;
2. Verifica se o número de bits 1 é ímpar. Se for, lança ValueError sinalizando detecção de erro;
3. Caso for par, não detecta erro e remove o último byte com o bit de paridade par do quadro e o retorna.

3.2.2 CRC

(I) DESCRIÇÃO GERAL

a. Técnica de EDC mais robusta que o *bit* de paridade par que envolve divisão polinomial do quadro de dados por um polinômio gerador G(x). Em suma, o processo consiste em:
   - Dividir o quadro pelo polinômio gerador G(x).
   - Adicionar o resto da divisão ao final do quadro, em um conjunto de bytes.

(J) IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE EDC

$$crc\big(quadro\colon bytes,\ tamanho_{do_{edc}}\colon int = 8,\ polinomio\colon int = 0x07\big) \rightarrow bytes$$

i. **Entrada:** Quadro sem o bit de paridade.
Ex: b"s"

ii. **Saída:** Quadro com o CRC.
Ex: b"s\x12\x34\x56\x78"

iii. Passo a passo:

1. Inicializa CRC como zero;

2. Divide quadro, em bytes, pelo polinômio gerador G(x);
3. Converte o valor final do CRC para bytes
4. Anexa o CRC ao final do quadro e retorna o quadro completo com o CRC.

b. FUNÇÃO DE VERIFICAÇÃO

$$verifica\_crc\left(quadro\colon bytes,\ tamanho_{do_{edc}}\colon int = 8,\ polinomio\colon int = 0x07\right) \rightarrow bytes$$

i. **Entrada:** Quadro com o CRC.
Ex: b"s\x12\x34\x56\x78"

ii. **Saída:** Quadro sem o bit de paridade.
Ex: b"s"

iii. Passo a passo:

1. Converte o tamanho do EDC (bits) para bytes, sabendo onde está o CRC.
2. Recalcula o EDC a partir do quadro recuperado e do polinômio informado (default=0x07)
3. Verifica o resultado do CRC:
   - Se o valor do CRC for zero, o quadro é válido;
   - Se o valor do CRC não for zero, lança ValueError indicando erro no quadro recebido;
4. Se for válido, remove os bytes do CRC e retorna o quadro original (sem o EDC).

3.3 PROTOCOLO DE CORREÇÃO DE ERROS

Técnicas de detecção e correção de erros são essenciais para garantir a integridade dos dados transmitidos, corrigindo falhas causadas por ruídos no meio físico. Entre os métodos existentes, utilizou-se o código de Hamming (7,4), que transforma cada grupo de 4 bits em 7 bits codificados, permitindo a correção automática de erros simples durante a transmissão.

3.3.1 Hamming (7,4)

A. DESCRIÇÃO GERAL

- O código de Hamming (7,4) transforma cada nibble (4 bits) em 7 bits codificados, inserindo 3 bits de paridade calculados de acordo com as seguintes operações XOR:

$$p1 = d1 \bigoplus d2 \bigoplus d4$$

$$p2 = d1 \bigoplus d3 \bigoplus d4$$

$$p3 = d2 \bigoplus d3 \bigoplus d4$$

- Posteriormente, constitui-se a palavra de código na forma:

$$palavra = [p1, p2, d1, p3, d2, d3, d4]$$

- Após a transmissão, verifica-se a integridade da palavra de código analisando as paridades internas dos dados recebidos. Esses valores de paridade interna $s1$, $s2$ *e* $s3$ indicam a posição do erro na palavra, que são percebidos apenas quando a síndrome é diferente de (000).

$$posição\ do\ erro = (s3\ s2\ s1)$$

$$s1 = p1 \bigoplus d1 \bigoplus d2 \bigoplus d4$$

$$s2 = p2 \bigoplus d1 \bigoplus d3 \bigoplus d4$$

$$s3 = p3 \bigoplus d2 \bigoplus d3 \bigoplus d4$$

(K)IMPLEMENTAÇÃO EM PYTHON

a. FUNÇÃO DE EDC

$$hamming(dado: bytes) \rightarrow bytes$$

i. **Entrada:** Um quadro, em *bytes*.
Ex: b"s" → 01110011

ii. **Saída:** Quadro com o hamming aplicado.
Ex: b"x\0f\x43"

iii. Passo a passo:

1. Para cada byte do quadro, divide-o em dois nibbles;
2. Para cada nibble, separa os bits de dados $d1$, $d2$, $d3$, $d4$ e calcula os bits de paridade $p1$, $p2$, $p3$;
3. Organiza os bits no formato Hamming(7,4);
4. Converte esse bloco de 7 bits em um byte (MSB é ignorado) e adiciona à lista de bytes codificados;

5. Retorna essa sequência de bytes após fazer isso em todos os nibbles.

b. FUNÇÃO DE VERIFICAÇÃO

$$verifica\_hamming(quadro\colon bytes) \rightarrow bytes$$

i. **Entrada:** Quadro com o hamming aplicado.
Ex: b”x\0f\x43”

ii. **Saída:** Um quadro, em *bytes*.
Ex: b”s”

iii. Passo a passo:

1. Percorre cada byte do quadro com EDC;
2. Separa os 7 bits úteis (removendo o MSB que foi inserido somente para completar 1 byte de tamanho);
3. Calcula a síndrome de erro usando os bits de paridade e os bits de dados;
4. Recupera os nibbles a partir dos bits corrigidos;
5. Após processar todos os bytes, combina cada par de nibbles para reconstruir os bytes originais do quadro;
6. Retorna o quadro original corrigido e decodificado.

4 MEMBROS

O projeto contou com a participação ativa de todos os integrantes. As participações no desenvolvimento da aplicação estão resumidas da seguinte forma:

- **Rodrigo Fonseca Torreao:** Implementou o esqueleto do projeto, o que inclui estrutura de diretórios e versões iniciais dos módulos. Contribuiu com a interface gráfica, transmissor, módulos utilitários, modulação ASK, enquadramento com FLAG Inserção de *bytes* e codificação Manchester.

- **Arthur Delpino Barbabella:** Implementou o receptor e a comunicação entre os dois módulos via *sockets*, modulação analógica FSK e 8-QAM, codificação NRZ-Polar, incremento de ruído, *bit* de paridade par, enquadramento com contagem de caracteres e contribuiu na construção dos gráficos no módulo utilitário.

- **João Victor Cavallin Pereira:** Implementou o Hamming, codificação bipolar, enquadramento com FLAGS Inserção de *bits* e contribuiu com o módulo de utilitários.

5 CONCLUSÃO

O desenvolvimento deste simulador integrado das camadas física e de enlace proporcionou uma compreensão prática de algum dos protocolos fundamentais em redes

de computadores. A implementação dos mecanismos de modulação digital e por portadora, enquadramento, detecção e correção de erros validou teoricamente os conceitos estudados.

Dentre as principais dificuldades encontradas no decorrer do trabalho, vale destacar a dificuldade em utilizar sockets na comunicação do receptor e transmissor por falta de familiaridade prévia, a aplicação das técnicas de modulação analógica, particularmente no que tange à obtenção de amostras úteis para plotagem em gráfico da matplotlib, e as manipulações com *bits* em *python* para implementar praticamente todos os métodos da camada de enlace.

6 CÓDIGO FONTE

- **Repositório do Projeto**