Universidade de Brasília - UnB
Departamento de Ciência da Computação

Disciplina: Teleinformática e Redes 1
Prof. Marcelo Antonio Marotta

Relatório

Simulação de Protocolos da Camada Física e de Enlace

Grupo: Os Transmissores

Membros:

Davi Sousa da Vinha	Matrícula: 232001667
Márcio Vieira dos Santos	Matrícula: 232029274

Brasília
2025

1 Introdução

A comunicação de dados moderna baseia-se em um complexo conjunto de camadas abstratas que garantem que a informação gerada por um usuário chegue íntegra ao seu destino, independentemente das imperfeições do meio físico. Este relatório documenta o desenvolvimento de um **Simulador Didático de Redes**, focado especificamente nas duas primeiras camadas do modelo OSI: a Camada Física e a Camada de Enlace de Dados.

O objetivo pedagógico deste projeto é desmistificar os processos "invisíveis"que ocorrem durante a transmissão. Em um cenário real, bits não trafegam de forma mágica; eles precisam ser organizados em estruturas lógicas (quadros), protegidos matematicamente contra ruídos e interferências, e convertidos em grandezas físicas (como tensão ou ondas).

O software desenvolvido simula este ciclo completo:

1. **Transmissão (Tx):** A conversão de texto (ASCII) para bits, o empacotamento lógico (Enquadramento), a adição de redundância para segurança (Controle de Erro) e a modulação em sinal analógico.
2. **Canal Ruidoso:** A simulação de um meio físico imperfeito através da injeção de Ruído Branco Gaussiano Aditivo (AWGN), permitindo testar a resiliência dos protocolos.
3. **Recepção (Rx):** O processo inverso de demodulação, sincronização, verificação de integridade e reconstrução da mensagem original.

2 Arquitetura e Lógica do Simulador

O sistema foi projetado em módulos independentes para refletir o encapsulamento real das redes. O núcleo do sistema é o módulo `simulador.py`, que atua como o "meio de comunicação", orquestrando a passagem de dados entre o Transmissor e o Receptor.

2.1 O Papel do Simulador

O `Simulador` não apenas conecta o transmissor ao receptor, mas também é responsável por introduzir a realidade física ao sistema: o ruído. Ele recebe um vetor de sinais "perfeitos"da camada física transmissora, aplica uma distorção matemática baseada em uma distribuição normal (Gaussiana), e entrega esse sinal "sujo"ao receptor. Isso permite testar a robustez dos algoritmos implementados.

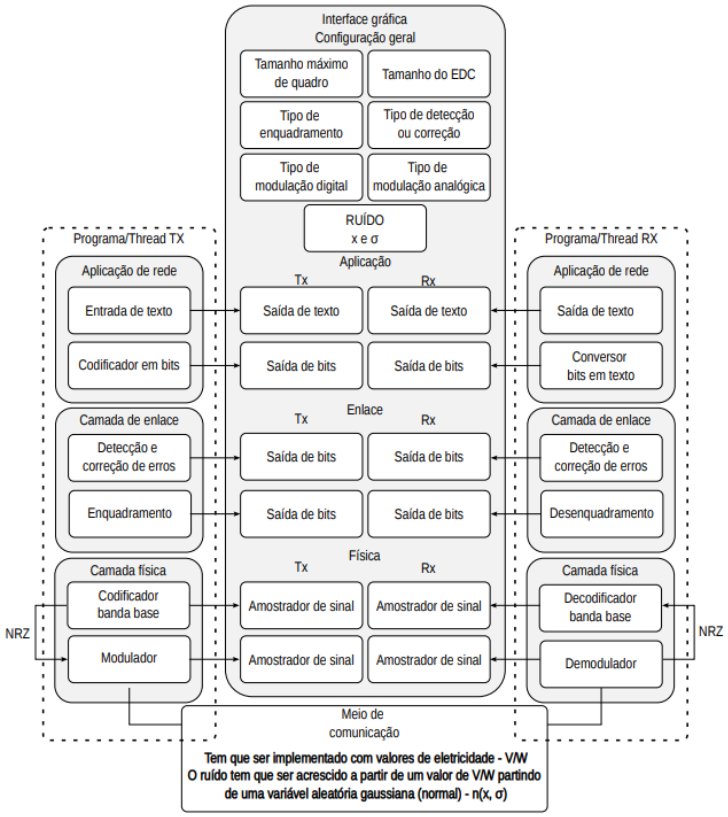

Figura 1: Diagrama de desenvolvimento

Figura 1: Diagrama de fluxo do simulador, destacando a transformação dos dados entre as camadas.

O fluxo de execução segue a lógica linear abaixo:

```
def executar_simulacao(self, mensagem, config):
    # 1. Camada de Aplicacao: Texto -> Bits
    bits_dados = self.bitamento(mensagem)

    # 2. Camada de Enlace (Tx):
    # Adiciona bits de protecao (ex: Hamming)
    # Adiciona cabecalho/flags (Enquadramento)

    # 3. Camada Fisica (Tx):
    # Transforma bits em vetor de sinais (ex: Voltagens)

    # 4. Meio de Comunicacao:
    # Adiciona ruido aleatorio (sigma) ao vetor
    sinal_com_ruido = self.ruido(sinal_modulado, config['sigma_ruido'])

    # 5. Camada Fisica (Rx):
    # Tenta recuperar os bits originais a partir do sinal ruidoso

```

```
19      # 6. Camada de Enlace (Rx):
20      # Remove flags e verifica se a matematica de erro bate
21
22      # 7. Camada de Aplicacao: Bits -> Texto
23      return status_final, texto_decodificado
```

Listing 1: Lógica principal do Simulador

3 Implementação da Camada de Enlace

A Camada de Enlace é responsável pela estruturação lógica. Abaixo detalhamos as implementações realizadas no arquivo `enlace.py`.

3.1 Enquadramento

1. Contagem de Caracteres: Calcula o número de bytes da mensagem e insere esse valor em um cabeçalho de 16 bits. *Lógica implementada:* O algoritmo exige que o dado seja um múltiplo exato de 8 bits (1 byte). Foi necessário implementar um **padding** (preenchimento com zeros) antes do cálculo do cabeçalho para evitar erros de arredondamento na divisão.

```
def enqCont(self, dados):
    padding = list(dados)
    # Padding essencial: Garante multiplo de 8 para formar bytes
   completos
    while len(padding) % 8 != 0:
        padding.append(0)

    quantbyt = len(padding) // 8
    header = self.trem(quantbyt, 16)
    return header + padding
```

2. Inserção de Bytes (Byte Stuffing): Utiliza um byte especial de `FLAG` (01111110) para delimitar o quadro. Para garantir a transparência (i.e., permitir que a flag apareça no meio dos dados sem fechar o quadro), usa-se um caractere de escape (`ESC`).

```
def enqBytes(self, dados):
    proteg = []
    for i in range(0, len(dados), 8):
        byta = dados[i : i+8]
        # Se o dado parece uma FLAG ou ESC, protege com ESC antes
        if byta == self.FLAG or byta == self.ESC:
            proteg.extend(self.ESC)
        proteg.extend(byta)
    return self.FLAG + proteg + self.FLAG
```

3. Inserção de Bits (Bit Stuffing): Uma técnica mais refinada que opera em nível de bit. A regra é simples: a sequência da flag (seis '1's) nunca deve aparecer nos dados. Para isso, o transmissor insere um '0' após qualquer sequência de cinco '1's.

```
def enqBits(self, dados):
    buffer_saida = []
    uns = 0
    for bit in dados:
```

```
        buffer_saida.append(bit)
        if bit == 1:
            uns += 1
            if uns == 5:
                buffer_saida.append(0) # Insere 0 de stuffing
                uns = 0
        else:
            uns = 0
    return self.FLAG + buffer_saida + self.FLAG
```

3.2 Controle de Erros (Detecção e Correção)

A física introduz ruído, que inverte bits (0 vira 1). A camada de enlace usa matemática para detectar ou corrigir isso.

1. Bit de Paridade Par: O método mais simples. Adiciona um bit extra para garantir que o número total de 1s seja sempre par. Detecta erros ímpares, mas falha se dois bits inverterem.

```
def detPar(self, dados):
    qtd_uns = dados.count(1)
    bit_paridade = 1 if qtd_uns % 2 != 0 else 0
    return dados + [bit_paridade]
```

2. Checksum (Soma de Verificação): Padrão da Internet (TCP/IP). Soma os dados em blocos de 16 bits e envia o complemento dessa soma. *Lógica implementada:* Assim como na contagem de caracteres, este algoritmo exige blocos de tamanho fixo (16 bits). Implementou-se um padding obrigatório para garantir que a mensagem seja divisível por 16 antes da soma. O *wrap around* garante que o estouro da soma (carry) não seja perdido.

```
def detSoma(self, dados):
    dados_copia = list(dados)
    # Padding obrigatorio para somar blocos de 16 bits
    while len(dados_copia) % 16 != 0:
        dados_copia.append(0)

    soma = 0
    for i in range(0, len(dados_copia), 16):
        # ... converte pedaco para int ...
        soma += pedaco_int
        while soma > 0xFFFF: # Wrap around (carry)
            carry = soma >> 16
            soma = soma & 0xFFFF
            soma += carry
    checksum = ~soma & 0xFFFF # Complemento de 1
```

```
    return dados + self.trem(checksum, 16)
```

3. CRC-32 (Verificação de Redundância Cíclica): Baseado em divisão polinomial. A mensagem é tratada como um polinômio gigante e dividida por um polinômio gerador padrão (IEEE 802.3). O resto da divisão é o CRC. É extremamente robusto contra erros em rajada.

```
def det32(self, dados):
    # Polinomio padrao IEEE 802.3
    crc = [1, 0, 0, ..., 1]
    # Adiciona 32 zeros ao final para o calculo
    dados_aumentados = list(dados) + [0] * 32

    for i in range(len(dados)):
        if dados_aumentados[i] == 1:
            for j in range(len(crc)):
                # Subtracao em GF(2) eh XOR
                dados_aumentados[i + j] ^= crc[j]

    resto = dados_aumentados[-(len(crc)-1):]
    return dados + resto
```

4. Código de Hamming (7,4): Diferente dos anteriores, este é um código de FEC (*Forward Error Correction*). Ele permite não apenas detectar, mas **corrigir** erros de 1 bit. O algoritmo trabalha com blocos de 4 bits de dados (chamados de **nibbles**). *Lógica implementada:* Como a mensagem original pode ter qualquer tamanho, foi necessário implementar um padding para garantir que o total de bits seja sempre múltiplo de 4, permitindo a divisão exata em nibbles.

```
def hamming(self, dados):
    dados_trabalho = list(dados)
    # Padding obrigatorio para dividir em nibbles (4 bits)
    while len(dados_trabalho) % 4 != 0:
        dados_trabalho.append(0)

    for i in range(0, len(dados_trabalho), 4):
        chunk = dados_trabalho[i : i+4]
        # ... (calculo das paridades p1, p2, p4) ...
        bloco_codificado = [p1, p2, d1, p4, d2, d3, d4]
        saida.extend(bloco_codificado)
    return saida
```

4 Implementação da Camada Física

A Camada Física (arquivo `fisica.py`) converte a sequência de bits em vetores de valores numéricos que representam o sinal analógico.

4.1 Codificação em Banda Base

Técnicas que alteram a voltagem direta da linha:

NRZ (Non-Return to Zero): Mapeia bit 1 para nível alto e 0 para nível baixo. Simples, mas sofre com perda de sincronia em longas sequências iguais.

```
def nrz(trem):
    sinal = []
    for bit in trem:
        if bit == 1: sinal.append(1)
        else:        sinal.append(-1)
    return sinal
```

Manchester: Resolve a sincronia inserindo uma transição de voltagem no meio de cada bit (XOR com o clock). O bit 1 vira [1, -1] (alto-baixo) e o bit 0 vira [-1, 1] (baixo-alto).

```
def manchester(trem):
    sinal = []
    for bit in trem:
        if bit == 1: sinal.extend([1, -1]) # Transicao descida
        else:        sinal.extend([-1, 1]) # Transicao subida
    return sinal
```

Bipolar (AMI): Utiliza três níveis de tensão. O nível zero representa o bit 0, e o bit 1 alterna entre positivo (+V) e negativo (-V), eliminando a componente contínua (DC).

```
def bipolar(trem):
    sinal = []
    ultimo = True
    for bit in trem:
        if bit == 1:
            if ultimo:
                sinal.append(1)
                ultimo = False
            else:
                sinal.append(-1)
                ultimo = True
        else:
            sinal.append(0)
    return sinal
```

4.2 Modulação por Portadora

Técnicas que modificam uma onda portadora senoidal para transmitir dados em canais como rádio ou fibra.

ASK e FSK: Modulam, respectivamente, a amplitude e a frequência da onda. São simples, mas menos eficientes espectralmente.

```
def ask(trem, amp):
    sinal = []
    for bit in trem:
        if bit == 1:
            # Gera 100 amostras de seno com amplitude definida
            for t in range(100):
                valor = amp * np.sin(2 * np.pi * (t / 100))
                sinal.append(valor)
        else:
            # Amplitude quase nula para bit 0
            for t in range(100): sinal.append(0.001)
    return sinal
```

QPSK (Quadrature Phase Shift Keying): Modula a fase da onda em 4 estados possíveis $(45°, 135°, 225°, 315°)$. Isso permite transmitir **2 bits por símbolo** (dibits). *Lógica implementada:* Para agrupar os bits em pares, foi necessário garantir que o vetor de entrada tivesse tamanho par, adicionando padding se necessário.

```
def qpsk(trem):
    # Padding: Garante numero par de bits para formar os pares
    if len(trem) % 2 != 0: trem.append(0)

    fase_map = {(0, 0): np.pi/4, (0, 1): 3*np.pi/4, ...}

    for i in range(0, len(trem), 2):
        b_pair = (trem[i], trem[i+1])
        fase = fase_map[b_pair] # Escolhe a fase pelo par de bits
        for n in range(100):
            valor = np.sin((n/100) + fase)
            sinal.append(valor)
    return sinal
```

16-QAM (Quadrature Amplitude Modulation): Combina amplitude e fase para criar 16 estados possíveis na constelação. Permite transmitir **4 bits por símbolo**, sendo extremamente eficiente. *Lógica implementada:* Similar ao QPSK, mas exigindo grupos de 4 bits. Um loop `while` foi adicionado para garantir o padding correto antes do mapeamento.

```
def qam16(trem):
    c4trem = list(trem)
```

```
    # Padding: Garante multiplo de 4 para formar quartetos
    while len(c4trem) % 4 != 0: c4trem.append(0)

    # Mapa da constelacao (Bit -> (Amplitude, Fase))
    map = {(0,0,0,0): (0.33, 225), ... }

    for i in range(0, len(c4trem), 4):
        quad = (c4trem[i], ..., c4trem[i+3])
        amp, fase_deg = map[quad]
        fase = fase_deg * np.pi / 180.0
        # Gera onda composta
        for n in range(100):
            valor = amp * np.sin(2 * np.pi * n/100 + fase)
            sinal.append(valor)
    return sinal
```

5 Membros e Divisão de Tarefas

O desenvolvimento foi colaborativo, com uma divisão clara de responsabilidades baseada nas camadas do modelo OSI, mas com forte integração na construção do núcleo do simulador.

- **Davi Sousa da Vinha (232001667):** Responsável pela arquitetura e implementação da **Camada Física**. Desenvolveu todos os algoritmos de modulação digital e por portadora (`fisica.py`) e suas contrapartes de demodulação (`decode_fis.py`). Foi essencial na lógica de renderização gráfica dos sinais e no tratamento matemático das ondas usando `numpy`.
- **Márcio Vieira dos Santos (232029274):** Responsável pela arquitetura e implementação da **Camada de Enlace**. Desenvolveu a lógica binária complexa, incluindo os algoritmos de enquadramento e toda a matemática de detecção e correção de erros (`enlace.py` e `decode_enlace.py`). No simulador, focou no tratamento lógico dos vetores de bits e padding.

A integração final no `simulador.py` e a construção da Interface Gráfica foram realizadas em conjunto (Pair Programming) para resolver os desafios de interconexão entre as camadas.

6 Conclusão e Desafios

O desenvolvimento deste simulador permitiu consolidar os conceitos teóricos vistos em sala de aula, demonstrando na prática que a comunicação de dados confiável é uma construção em camadas, onde cada etapa adiciona robustez ou eficiência ao processo.

6.1 O Desafio do Alinhamento de Bits (Padding)

A maior dificuldade técnica enfrentada durante a integração das camadas não foi um algoritmo específico, mas sim a necessidade sistemática de **alinhamento de bits (Padding)** entre diferentes protocolos que operam com granularidades distintas.

Observou-se que quase todas as etapas do processo exigiam que o fluxo de bits tivesse um tamanho específico para funcionar corretamente:

- O **Código de Hamming** exige blocos exatos de 4 bits (nibbles) para calcular a matriz de paridade.
- O protocolo de **Checksum** e a **Contagem de Caracteres** operam sobre blocos de 16 bits ou 8 bits (bytes).
- As modulações **QPSK** e **QAM-16** exigem, respectivamente, grupos de 2 e 4 bits para formar seus símbolos.

Como a mensagem do usuário (e os bits de controle adicionados, como CRC) pode ter qualquer comprimento arbitrário, isso gerava conflitos constantes.

Solução: A solução consistiu em aplicar padding defensivo em todas as funções de transmissão (como demonstrado nos códigos acima) (`EnlaceRx`).