Projeto - TR1

Simulador de camada física e de enlace

Turma 01

Giovanni Minari Zanetti - 202014280
João Pedro de Souza Lopes - 221006342
Eduardo Augusto Volpi - 190134330

Sumário

1 Introdução **2**

2 Implementação **2**
2.1 Camada Física 2
2.1.1 Modulação Digital 2
2.1.2 Modulação por Portadora 2
2.2 Camada de Enlace 3
2.2.1 Enquadramento de Dados 3
2.2.2 Detecção de Erros 3
2.2.3 Correção de Erros 4

3 Membros **5**
3.1 João Pedro de Souza Lopes 5
3.1.1 Interface 5
3.1.2 integração 6
3.1.3 testes 6
3.2 Giovanni Minari Zanetti 8
3.2.1 Modulação Digital 8
3.2.2 Modulação Digital 10
3.3 Eduardo Augusto Volpi 17
3.3.1 Camada de Enlace 17
3.3.2 Enquadramento 17
3.3.3 Detecção de Erros 17
3.3.4 Correção de Erros 18
3.3.5 Transmissão e Recepção 18

4 Conclusão **18**

5 Referências bibliográficas **19**

1 Introdução

Este relatório apresenta a implementação de um simulador que abrange as camadas física e de enlace do modelo OSI (Open Systems Interconnection). O projeto visa simular o funcionamento dessas camadas cruciais na comunicação de dados, focando em técnicas de modulação digital, modulação por portadora, enquadramento de dados, detecção e correção de erros. Esta abordagem abrangente proporciona uma compreensão profunda dos mecanismos fundamentais que garantem a transmissão eficiente e confiável de dados em redes de computadores.

2 Implementação

O simulador foi implementado em Python, abrangendo dois principais componentes: a camada física e a camada de enlace. Cada componente foi desenvolvido com atenção aos detalhes específicos dos protocolos e técnicas relevantes.

2.1 Camada Física

A implementação da camada física focou em duas categorias principais de modulação:

2.1.1 Modulação Digital

Foram implementados três tipos de modulação digital:

- **Non-return to Zero Polar (NRZ-Polar)**: Esta técnica representa os bits '0' e '1' com níveis de tensão opostos, mantendo o nível constante durante o intervalo de bit. A implementação envolveu a criação de uma função que mapeia os bits de entrada para os níveis de tensão correspondentes.
- **Manchester**: Nesta codificação, cada bit é representado por uma transição. Um '0' é codificado como uma transição de positivo para negativo, enquanto um '1' é codificado como uma transição de negativo para positivo. A implementação necessitou de uma lógica para gerar essas transições para cada bit de entrada.
- **Bipolar**: Esta técnica alterna entre polaridades positiva e negativa para representar '1's, enquanto '0's são representados por ausência de sinal. A implementação manteve um estado interno para alternar corretamente as polaridades dos '1's.

2.1.2 Modulação por Portadora

Foram implementados três tipos de modulação por portadora:

- **Amplitude Shift Keying (ASK)**: Esta técnica varia a amplitude da onda portadora para representar diferentes símbolos. A implementação envolveu a geração de uma onda senoidal com amplitudes variáveis baseadas nos bits de entrada.
- **Frequency Shift Keying (FSK)**: Nesta técnica, a frequência da onda portadora é alterada para representar diferentes símbolos. A implementação requeriu a geração de ondas senoidais com frequências distintas para '0's e '1's.
- **8-Quadrature Amplitude Modulation (8-QAM)**: Esta técnica combina modulação de amplitude e fase para transmitir 3 bits por símbolo. A implementação envolveu o mapeamento de grupos de 3 bits para 8 estados diferentes de amplitude e fase da onda portadora.

2.2 Camada de Enlace

A implementação da camada de enlace foi dividida em três componentes principais: enquadramento de dados, detecção de erros e correção de erros. Cada componente foi desenvolvido de forma modular para facilitar testes e manutenção.

2.2.1 Enquadramento de Dados

Foram implementados dois métodos de enquadramento: contagem de caracteres e inserção de bytes.

Contagem de Caracteres Este método prefixou cada quadro com um campo de contagem de 4 bytes que indica o número de caracteres no quadro. Procedimento de enquadramento:

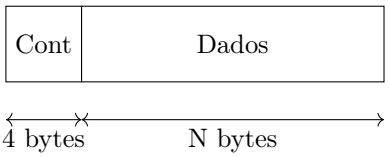


Figura 1: Estrutura do quadro com contagem de caracteres

Calcular o tamanho dos dados em bytes. Converter o tamanho para um inteiro de 4 bytes. Concatenar o tamanho com os dados.

Procedimento de desenquadramento:

Ler os primeiros 4 bytes para obter o tamanho. Converter os 4 bytes para um inteiro. Ler o número de bytes especificado para obter os dados.

Inserção de Bytes Este método utiliza bytes de flag especiais (0x7E) para delimitar o início e o fim de cada quadro, com um mecanismo de escape (0x7D) para tratar ocorrências desses bytes especiais nos dados. Procedimento de enquadramento:

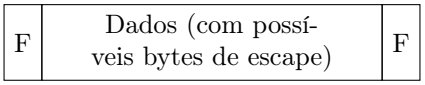


Figura 2: Estrutura do quadro com inserção de bytes

Iniciar o quadro com o byte de flag (0x7E). Para cada byte dos dados: a. Se o byte for 0x7E ou 0x7D, inserir um byte de escape (0x7D) seguido pelo byte original XOR 0x20. b. Caso contrário, inserir o byte original. Finalizar o quadro com o byte de flag (0x7E).

Procedimento de desenquadramento:

Remover o byte de flag inicial. Para cada byte dos dados: a. Se o byte for 0x7D, ler o próximo byte e fazer XOR com 0x20. b. Caso contrário, manter o byte original. Remover o byte de flag final.

Decisão de implementação: Optamos por usar um byte de escape em vez de duplicação de bytes para lidar com ocorrências de bytes de flag nos dados, pois isso reduz a expansão do quadro em casos onde há muitas ocorrências de bytes de flag.

2.2.2 Detecção de Erros

Foram implementados dois métodos de detecção de erros: bit de paridade par e CRC-32.

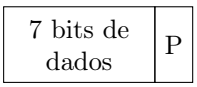


Figura 3: Estrutura de um byte com bit de paridade

Bit de Paridade Par Este método adiciona um bit extra a cada byte para garantir que o número total de bits '1' seja par. Procedimento de adição de paridade:

Para cada byte de dados: a. Contar o número de bits '1'. b. Se o número for ímpar, adicionar um bit '1' como paridade. c. Se o número for par, adicionar um bit '0' como paridade.

Procedimento de verificação de paridade:

Para cada byte recebido: a. Contar o número total de bits '1' (incluindo o bit de paridade). b. Se o número for ímpar, sinalizar erro.

CRC-32 (IEEE 802) Este método adiciona um valor de verificação de 32 bits ao final de cada quadro, calculado usando o polinômio CRC-32 padrão. Procedimento de cálculo e adição do

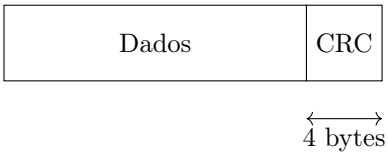


Figura 4: Estrutura do quadro com CRC-32

CRC:

Inicializar o registrador CRC com 0xFFFFFFFF. Para cada byte dos dados: a. Fazer XOR do byte com o byte menos significativo do registrador CRC. b. Para cada bit do resultado: i. Se o bit menos significativo do registrador CRC for 1, deslocar o registrador à direita e fazer XOR com o polinômio 0xEDB88320. ii. Caso contrário, apenas deslocar o registrador à direita. Fazer o complemento do registrador CRC. Adicionar o valor do CRC ao final dos dados.

Procedimento de verificação do CRC:

Calcular o CRC dos dados recebidos (incluindo o CRC anexado). Se o resultado for 0, os dados estão íntegros.

Decisão de implementação: Utilizamos a biblioteca zlib para o cálculo do CRC-32, pois ela oferece uma implementação eficiente e bem testada.

2.2.3 Correção de Erros

Para correção de erros, implementamos o código de Hamming.

Código de Hamming Este código adiciona bits de paridade extras que permitem a detecção e correção de erros de um único bit. Procedimento de codificação:

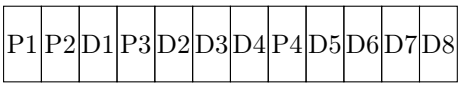


Figura 5: Estrutura de um código de Hamming (12,8)

Posicionar os bits de dados nas posições que não são potências de 2. Calcular os bits de paridade: P1 = D1 $\oplus$ D2 $\oplus$ D4 $\oplus$ D5 $\oplus$ D7 P2 = D1 $\oplus$ D3 $\oplus$ D4 $\oplus$ D6 $\oplus$ D7 P3 = D2 $\oplus$ D3 $\oplus$ D4 $\oplus$ D8 P4 = D5 $\oplus$ D6 $\oplus$ D7 $\oplus$ D8 Inserir os bits de paridade nas posições de potência de 2.

Procedimento de decodificação e correção:

Calcular os bits de síndrome: S1 = P1 ⊕ D1 ⊕ D2 ⊕ D4 ⊕ D5 ⊕ D7 S2 = P2 ⊕ D1 ⊕ D3 ⊕ D4 ⊕ D6 ⊕ D7 S3 = P3 ⊕ D2 ⊕ D3 ⊕ D4 ⊕ D8 S4 = P4 ⊕ D5 ⊕ D6 ⊕ D7 ⊕ D8 Se todos os bits de síndrome forem 0, não há erro. Caso contrário, o valor binário formado pelos bits de síndrome indica a posição do bit com erro. Inverter o bit na posição indicada para corrigir o erro.

Decisão de implementação: Implementamos o código de Hamming (12,8), que permite codificar 8 bits de dados em 12 bits, oferecendo um bom equilíbrio entre capacidade de correção e overhead.

3 Membros

3.1 João Pedro de Souza Lopes

O participante foi responsavel pela maior parte da interface ,integração,e teste do trabalho

3.1.1 Interface

A interface foi feita em tkinter e consiste em um quadro dividido em 4 partes sendomutilizadas da seguinte forma :

-(acima e a esquerda) esta seção mostra a forma do barulho do meio setado

-(acima e a direita) esta seção mostra o painel de controle com opções de tipos de bitstream

-(abaixo e a esquerda) esta seção mostra a forma inadulterada pelo barulho do meio da onda enviada

-(abaixo e a direita)esta seção mostra a onda como deve ser recebida

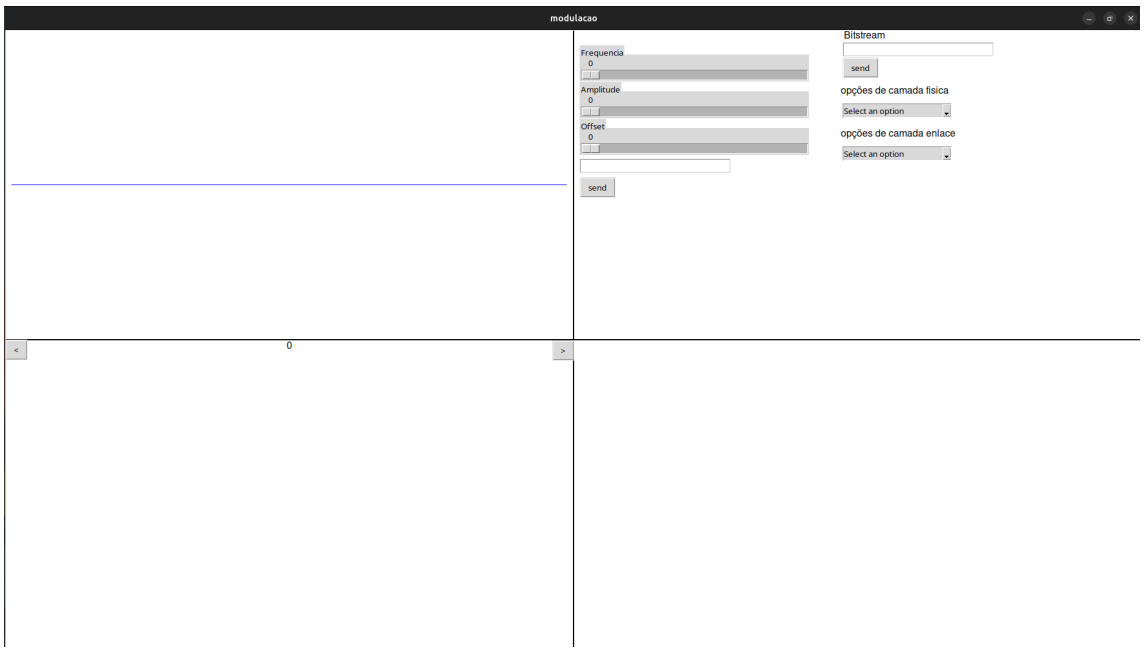


Figura 6: interface inicial.

3.1.2 integração

A integração foi feita de forma que as funções de enlace recebessem uma string contendo os bits e devolvessem uma string de bits para o protocolo escolhido da camada fisica que deveria devolver uma lista com os valores que sao repassados para a paginação reformatação e enfim desenhados

as seguintes funções foram mudadas para se conformar com o modelo acima apresentado:8-qam,crc,ask,fsk

```
if ProtocoloEnlace == "Contagem de Caracteres":
    bitMessage=Enlace.contagem_de_char(bitMessage)
elif ProtocoloEnlace == "Intersecção de bytes":
    bitMessage=Enlace.enquadrar_insercao_bytes(bitMessage)
elif ProtocoloEnlace == "Bit de paridade":
    bitMessage=Enlace.adicionar_bit_deParidade(bitMessage)
elif ProtocoloEnlace == "CRC":
    bitMessage=Enlace.crc16(bitMessage)
elif ProtocoloEnlace == "Hamming":
    bitMessage=Enlace.adicionar_hamming(bitMessage)
    n=1
Fisica=camadaFisica()
if ProtocoloFisica == "NRZ-Polar":
    wave=Fisica.nrz_polar(bit_stream=bitMessage)
elif ProtocoloFisica == "Manchester":
    wave=Fisica.manchester(bit_stream=bitMessage)
elif ProtocoloFisica == "Bipolar":
    wave=Fisica.bipolar(bit_stream=bitMessage)
elif ProtocoloFisica == "ASK":
    wave=Fisica.ask(dig_signal=bitMessage,h=n)
elif ProtocoloFisica == "FSK":
    wave=Fisica.fsk(dig_signal=bitMessage,h=n)
elif ProtocoloFisica == "8-QAM":
    wave=Fisica.qam8_modulation(dig_signal=bitMessage)
Wave = []
```

Figura 7: integração.

3.1.3 testes

Foram feitos testes de tipos e tamanhos variados de todas as funções e foram aplicados diferentes multiplicadores a fim de facilitar a visualização das ondas e a aferição de suas caracteristicas alem de eliminar a maior quantidade possivel de warnings e erros (mesmo que eles não prejudicassem o funcionamento do programa alem disso o programa passou por refatoração e redimensionamento por mim com a ajuda do pylint para evitar redundancia e bloatware

```
Sum=False
alredy=False
offset=""
freq=0
for i in range(len(x)):

    if x[i].isnumeric() and openbrackets==False:
        if amp!=0:
            amp=(10*amp)+int(x[i])
            if x[i-2]=="-":
                amp=amp*-1
        else:
            amp=int(x[i])
            if i!=0:
                if x[i-1]=="-" and x[i+1].isnumeric()!=True:
                    amp=amp*-1
    elif x[i]=="c":
        if amp==0:
            amp=1
        function="cos"
    elif x[i]=="e":
        if amp==0:
            amp=1
        function="sen"
    elif x[i]=="(":
```

Figura 8: dicas do pylint.

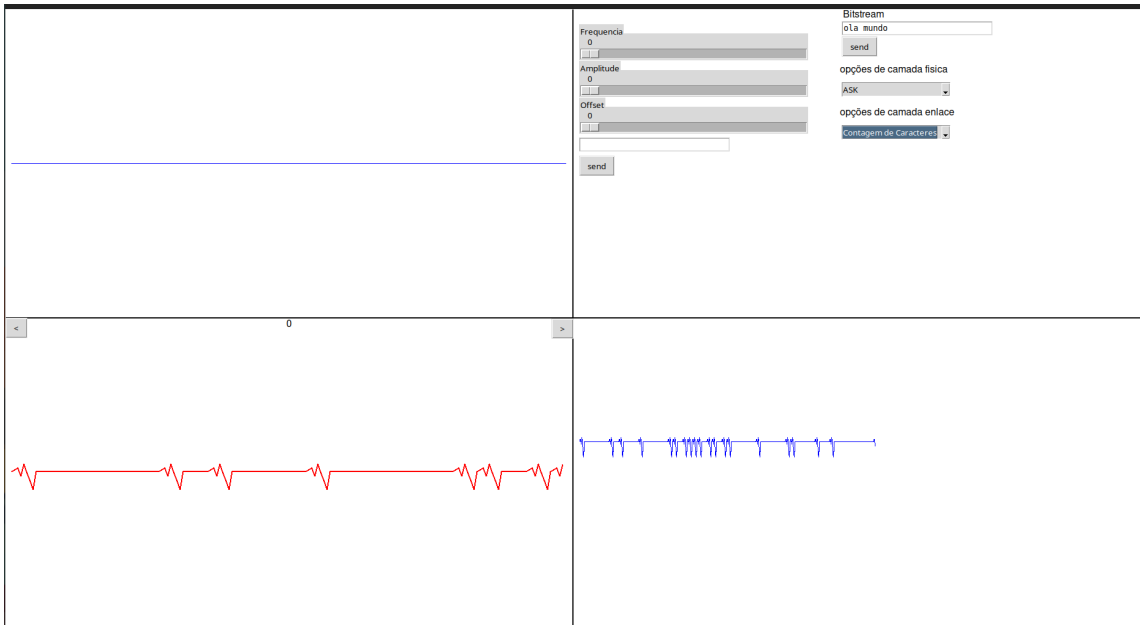


Figura 9: teste de deste dado.

3.2 Giovanni Minari Zanetti

O aluno foi responsável pela implementação da camada física e implementou diversas técnicas de modulação digital e modulação por portadora em Python. O código desenvolvido abrange as modulações digitais NRZ polar, Manchester e Bipolar, e as modulações por portadora ASK, FSK, e 8-QAM, fundamentais para a comunicação digital. A seguir, é descrito o funcionamento de cada uma dessas técnicas.

3.2.1 Modulação Digital

A modulação digital é a técnica que transforma uma sequência de bits (bits 0 e 1) em sinais que podem ser transmitidos por um canal de comunicação. No código desenvolvido, as seguintes técnicas foram implementadas:

NRZ Polar (Non-Return-to-Zero Polar): A função *nrz_polar* converte uma sequência de bits em um sinal digital onde o bit 1 é mapeado para o valor 1, e o bit 0 para o valor -1. Este tipo de modulação é simples e eficaz, mas pode ser suscetível a problemas de sincronização em longas sequências de bits iguais.

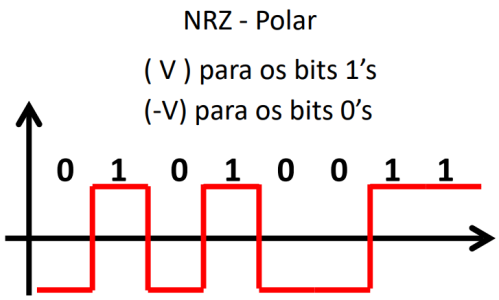


Figura 10: Esquemático do NRZ Polar.

A figura 10 mostra o comportamento do sinal digital gerado por dado bit stream a partir do NRZ Polar.

O código a seguir contém a implementação da modulação, que recebe um bit stream, e retorna um sinal digital, que pode ser usado em uma modulação por portadora.

```
def nrz_polar(self,bit_stream):
        #Se o bit for 1, sinal digital correspondente é 1, e se for 0, sinal
        ↪  digital correspondente é -1
        dig_signal = []
        for bit in bit_stream:
            if bit == "1":
                dig_signal.append(1)
            else:
                dig_signal.append(-1)
        return dig_signal
```

Manchester: A função *manchester* implementa a modulação Manchester, onde cada bit é representado por dois valores: um pulso positivo seguido de um pulso negativo para o bit 1, e um pulso negativo seguido de um pulso positivo para o bit 0. Esta técnica facilita a sincronização do sinal, uma vez que há sempre uma transição no meio de cada bit.

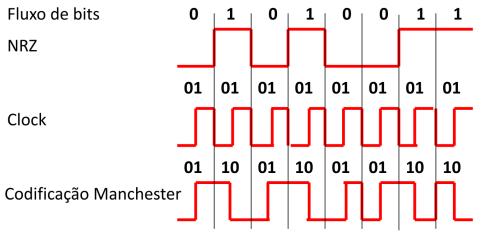


Figura 11: Esquemático da codificação Manchester.

A figura 11 mostra o comportamento do sinal digital gerado por dado bit stream a partir da codificação Manchester.

O código a seguir contém a implementação da modulação, que recebe um bit stream, e retorna um sinal digital, que pode ser usado em uma modulação por portadora:

```
def manchester(self,bit_stream):
        #Combina clock com o bit atual a partir de operação xor
        dig_signal = []
        for bit in bit_stream:
            if bit == "0":
                #Faz o xor entre o clock e o bit atual (0)
                dig_signal.append(0)
                dig_signal.append(1)
            else:
                #Faz o xor entre o clock e o bit atual (1)
                dig_signal.append(1)
                dig_signal.append(0)
        return dig_signal
```

A função realiza o xor entre o clock e cada bit do bit stream.

Bipolar: A função *bipolar* realiza a modulação bipolar, na qual o bit 0 é mapeado para o valor 0, e o bit 1 alterna entre 1 e -1. Esta alternância reduz a componente DC no sinal transmitido, sendo vantajosa em alguns sistemas de comunicação.

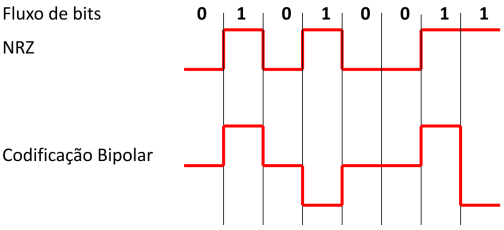


Figura 12: Esquemático da codificação Bipolar.

A figura 12 mostra o comportamento do sinal digital gerado por dado bit stream a partir da codificação Bipolar.

O código a seguir contém a implementação da modulação, que recebe um bit stream, e retorna um sinal digital, que pode ser usado em uma modulação por portadora:

```
def bipolar(self,bit_stream):
        #Se o bit for 0, sinal digital correspondente é 0, e se for 1, alterna
        ↪  entre 1 e -1
        dig_signal = []
        flag = 0
```

```
        for bit in bit_stream:
            if bit == "1":
                if flag == 0:
                    dig_signal.append(1)
                    flag = 1
                else:
                    dig_signal.append(-1)
                    flag = 0
            else:
                dig_signal.append(0)
        return dig_signal
```

A função conta com uma flag que sinaliza se o sinal anterior foi 1 ou -1, então o sinal digital alterna entre 1 e -1 caso o bit atual do bit stream seja 1, e é 0 caso o bit atual seja 0.

3.2.2 Modulação Digital

A modulação por portadora consiste em utilizar uma onda portadora de alta frequência para transportar o sinal digital, facilitando sua transmissão por meios físicos, como cabos ou sinais de rádio. No código, foram implementadas as seguintes técnicas:

ASK (Amplitude Shift Keying): A função *ask* realiza a modulação ASK, onde a amplitude da portadora é modificada conforme o bit do sinal digital. Para o bit 1, a portadora é transmitida com amplitude total, enquanto para o bit 0 ou -1, a amplitude é zero. Este tipo de modulação é simples, mas pode ser sensível ao ruído.

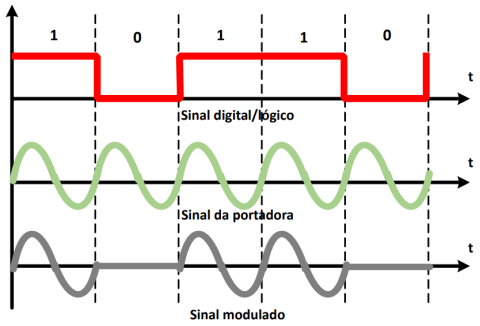


Figura 13: Esquemático da modulação ASK.

A figura 13 mostra um exemplo do comportamento do sinal gerado por dado sinal digital a partir da modulação ASK.

O código a seguir contém a implementação da modulação, que recebe um sinal digital, e retorna um sinal da portadora:

```
def ask(self, dig_signal, f=10000, sample=1000,h=0):
        if len(dig_signal)==0:
            return []
        if len(dig_signal)>=400:
            Signal=[]
            part_length = len(dig_signal) // 6

            # Calculate any remaining characters after dividing into 4 parts
            remainder = len(dig_signal) % 6

            # Initialize a list to store the 4 parts
```

```
        parts = []
        start = 0

        for i in range(6):
            # If there are remaining characters, add one extra to the
            ↪  current part
            end = start + part_length + (1 if i < remainder else 0)

            # Append the current part to the list of parts
            parts.append(dig_signal[start:end])

            # Update the start index for the next part
            start = end
        for i in range(len(parts)):
            dig_signal=parts[i]
            #Calcula o número de amostras por bit, baseado no número total
            ↪  de amostras e no comprimento do sinal digital
            samples_per_bit = sample // len(dig_signal)
            signal = []
            if h==1:
                n=100
            else:
                n=1
            #Modula a onda portadora com base nos bits de dados
            for bit in dig_signal:
                for j in range(samples_per_bit):
                    #Calcula o tempo atual como uma fração do número total
                    ↪  de amostras
                    time = j / sample

                    #Se o bit é 1, adiciona uma onda senoidal à lista de
                    ↪  sinais
                    if bit == 1:
                        signal.append(sin(2 * pi * f *
                        ↪  time)*19000000000*len(dig_signal)*n)
                    else:
                        #Se o bit é 0, adiciona zero à lista de sinais
                        ↪  (amplitude zero para o bit 0)
                        signal.append(0)
            Signal=Signal+signal
        return Signal

    else:
        #Calcula o número de amostras por bit, baseado no número total de
        ↪  amostras e no comprimento do sinal digital
        samples_per_bit = sample // len(dig_signal)
        signal = []

        #Modula a onda portadora com base nos bits de dados
        for bit in dig_signal:
            for j in range(samples_per_bit):
```

```
            #Calcula o tempo atual como uma fração do número total de
            ↪  amostras
            time = j / sample

            #Se o bit é 1, adiciona uma onda senoidal à lista de
            ↪  sinais
            if bit == 1:
                signal.append(sin(2 * pi * f *
                ↪  time)*190000000000*len(dig_signal))
            else:
                #Se o bit é 0, adiciona zero à lista de sinais
                ↪  (amplitude zero para o bit 0)
                signal.append(0)

        #Retorna o sinal modulado
        return signal
```

A função recebe como parâmetros o sinal digital, uma frequência da portadora, que por default é 10kHZ, uma quantidade de amostras para o sinal, que por default é 1000 e um parâmetro h, que identifica se o bit_stream possui código de Hamming. Como dependendo da modulação digital, o tamanho do sinal digital é diferente, calcula-se a quantidade de amostras por "bit", baseado na quantidade total de amostras e do tamanho do sinal, sendo o resultado um número inteiro. Caso o "bit"atual seja 1, o valor atual do sinal é calculado utilizando a função seno. Caso contrário, a amplitude é 0.

FSK (Frequency Shift Keying): A função *fsk* implementa a modulação FSK, onde a frequência da portadora varia de acordo com o bit transmitido. Um bit 1 é representado por uma frequência mais alta, enquanto um bit 0 é representado por uma frequência mais baixa. Esta modulação é mais robusta ao ruído do que a ASK, pois a informação é codificada em frequência.

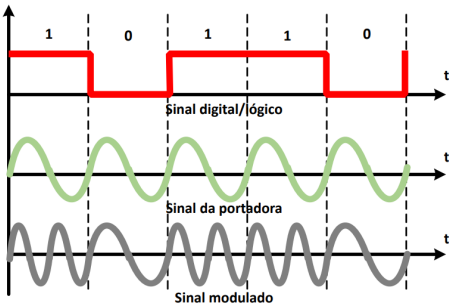


Figura 14: Esquemático da modulação FSK.

A figura 14 mostra um exemplo do comportamento do sinal gerado por dado sinal digital a partir da modulação FSK.

O código a seguir contém a implementação da modulação, que recebe um sinal digital, e retorna um sinal da portadora:

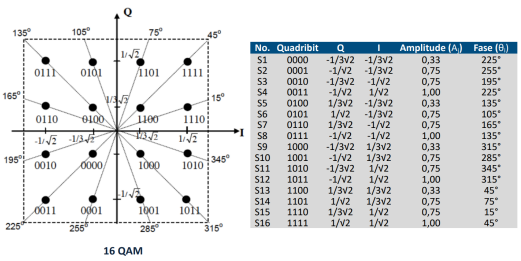
```
def fsk(self,dig_signal, f1=1000, f2=10000, sample=1000,h=0):
        if len(dig_signal)==0:
            return []
        if len(dig_signal)>100:
            Signal=[]
            part_length = len(dig_signal) // 6
```

```
        # Calculate any remaining characters after dividing into 4 parts
        remainder = len(dig_signal) % 6

        # Initialize a list to store the 4 parts
        parts = []
        start = 0

        for i in range(6):
            # If there are remaining characters, add one extra to the
            ↪  current part
            end = start + part_length + (1 if i < remainder else 0)

            # Append the current part to the list of parts
            parts.append(dig_signal[start:end])

            # Update the start index for the next part
            start = end
        for i in range(len(parts)):
            dig_signal=parts[i]
            #Calcula o número de amostras por bit, baseado no número total
            ↪  de amostras e no comprimento do sinal digital
            samples_per_bit = sample // len(dig_signal)
            signal = []
            if h==1:
                n=10
            else:
                n=1
            #Itera sobre cada bit no sinal digital
            for bit in dig_signal:
                #Para cada bit, gera uma sequência de amostras
                for j in range(samples_per_bit):
                    #Calcula o tempo atual como uma fração do número total
                    ↪  de amostras
                    time = j / sample

                    #Se o bit é 1, usa a frequência f2 para gerar a onda
                    ↪  senoidal
                    if bit == 1:
                        signal.append(sin(2 * pi * f2 *
                        ↪  time)*190000000000*len(dig_signal)*n)
                    else:
                        #Se o bit é 0, usa a frequência f1 para gerar a
                        ↪  onda senoidal
                        signal.append(sin(2 * pi * f1 *
                        ↪  time)*190000000000*len(dig_signal)*n)
            Signal=Signal+signal
        return Signal

    else:
        #Calcula o número de amostras por bit, baseado no número total de
        ↪  amostras e no comprimento do sinal digital
```

```
    samples_per_bit = sample // len(dig_signal)
    signal = []

    #Itera sobre cada bit no sinal digital
    for bit in dig_signal:
        #Para cada bit, gera uma sequência de amostras
        for j in range(samples_per_bit):
            #Calcula o tempo atual como uma fração do número total de
            ↪  amostras
            time = j / sample

            #Se o bit é 1, usa a frequência f2 para gerar a onda
            ↪  senoidal
            if bit == 1:
                signal.append(sin(2 * pi * f2 *
                ↪  time)*170000000000*len(dig_signal))
            else:
                #Se o bit é 0, usa a frequência f1 para gerar a onda
                ↪  senoidal
                signal.append(sin(2 * pi * f1 *
                ↪  time)*170000000000*len(dig_signal))

    #Retorna o sinal modulado
    return signal
```

A função recebe como parâmetros o sinal digital, duas frequências da portadora, que por default são 1kHZ e 10kHz, uma quantidade de amostras para o sinal, que por default é 1000 e o parâmetro h, que identifica se o bit_stream possui código de Hamming. Calcula-se a quantidade de amostras por "bit", baseado na quantidade total de amostras e do tamanho do sinal, sendo o resultado um número inteiro, assim como na ASK. Caso o "bit"atual seja 1, o valor atual do sinal é calculada utilizando a frequência 2. Caso contrário, o valor atual do sinal é calculada utilizando a frequência 1.

8-QAM (8-Quadrature Amplitude Modulation): A função *qam8_ modulation* implementa a modulação 8-QAM, uma técnica mais avançada onde grupos de três bits são mapeados para símbolos em um plano I-Q, permitindo a transmissão de mais bits por símbolo. A combinação de amplitude e fase permite uma maior eficiência espectral, sendo amplamente utilizada em sistemas modernos de comunicação.



No.	Quadribit	Q	I	Amplitude (A.)	Fase (θ.)
S1	0000	-1/3√2	-1/3√2	0,33	225°
S2	0001	-1/√2	-1/3√2	0,75	255°
S3	0010	-1/3√2	-1/√2	0,75	195°
S4	0011	-1/√2	1/√2	1,00	225°
S5	0100	1/3√2	-1/3√2	0,33	135°
S6	0101	1/√2	-1/3√2	0,75	105°
S7	0110	1/3√2	-1/√2	0,75	165°
S8	0111	-1/√2	-1/√2	1,00	135°
S9	1000	-1/3√2	1/3√2	0,33	315°
S10	1001	-1/√2	1/3√2	0,75	285°
S11	1010	-1/3√2	1/√2	0,75	345°
S12	1011	-1/√2	1/√2	1,00	315°
S13	1100	1/3√2	1/3√2	0,33	45°
S14	1101	1/√2	1/3√2	0,75	75°
S15	1110	1/3√2	1/√2	0,75	15°
S16	1111	1/√2	1/√2	1,00	45°

Figura 15: Esquemático da modulação 16-QAM.

A figura 15 mostra um exemplo de modulação 16-QAM, que tem uma lógica parecida com o 8-QAM. Ao invés de usar os símbolos apenas para modular o sinal em fase, também se modula em amplitude.

O código a seguir contém a implementação da modulação, que recebe um sinal digital, e

retorna um sinal da portadora:

```
    def qam_mapping(self,dig_signal):
        #Define o mapeamento dos bits para símbolos QAM (8-QAM neste caso)

        symbol_map = {
            (0, 0, 0): (1, 1),
            (0, 0, 1): (1, -1),
            (0, 1, 0): (-1, 1),
            (0, 1, 1): (-1, -1),
            (1, 0, 0): (1/3, 1/3),
            (1, 0, 1): (1/3, -1/3),
            (1, 1, 0): (-1/3, 1/3),
            (1, 1, 1): (-1/3, -1/3)
        }
        symbols = []
        #Itera sobre o sinal digital em blocos de 3 bits
        for i in range(0, len(dig_signal), 3):
            #Obtém um bloco de 3 bits e mapeia para o símbolo correspondente
            bits = tuple(dig_signal[i:i + 3])
            symbols.append(symbol_map[bits])
        return symbols

    def qam8_modulation(self,dig_signal, sample=500):
        if len(dig_signal)>500:
            Signal=[]
            part_length = len(dig_signal) // 6

            # Calculate any remaining characters after dividing into 4 parts
            remainder = len(dig_signal) % 6

            # Initialize a list to store the 4 parts
            parts = []
            start = 0

            for i in range(6):
                # If there are remaining characters, add one extra to the
                ↪  current part
                end = start + part_length + (1 if i < remainder else 0)

                # Append the current part to the list of parts
                parts.append(dig_signal[start:end])

                # Update the start index for the next part
                start = end
            for i in range(len(parts)):
                dig_signal=parts[i]
                if (int(len(dig_signal))%3!=0):
                    while (int(len(dig_signal))%3!=0):
                        dig_signal.append(0)
                        print("numero nao multiplo de 3")
                #Troca -1 por 0 para permitir o mapeamento na constelação QAM
                dig_signal = [0 if x == -1 else x for x in dig_signal]
```

```
            #Mapeia o sinal digital para símbolos QAM usando a função
             ↪  qam_mapping
            symbols = self.qam_mapping(dig_signal)
            if symbols!=[]:
            #Calcula o número de amostras por símbolo
                samples_per_symbol = sample // len(symbols)
            else:
                samples_per_symbol =0
            signal = []
            #Gera o sinal modulador para cada símbolo
            for symbol in symbols:
                I, Q = symbol
                for j in range(samples_per_symbol):
                    #Calcula o tempo atual como uma fração do número total
                     ↪  de amostras
                    time = j / sample
                    #Calcula o valor do sinal modulador para a amostra
                     ↪  atual
                    signal.append(I * cos(2 * pi * time) + Q * sin(2 * pi
                     ↪  * time))
            Signal=Signal+signal
        return Signal

    else:
        if (int(len(dig_signal))%3!=0):
            while (int(len(dig_signal))%3!=0):
                dig_signal.append(0)
                print("numero nao multiplo de 3")
        #Troca -1 por 0 para permitir o mapeamento na constelação QAM
        dig_signal = [0 if x == -1 else x for x in dig_signal]

        #Mapeia o sinal digital para símbolos QAM usando a função
         ↪  qam_mapping
        symbols = self.qam_mapping(dig_signal)
        if symbols!=[]:
        #Calcula o número de amostras por símbolo
            samples_per_symbol = sample // len(symbols)
        else:
            samples_per_symbol = 0
        signal = []

        #Gera o sinal modulador para cada símbolo
        for symbol in symbols:
            I, Q = symbol
            for j in range(samples_per_symbol):
                #Calcula o tempo atual como uma fração do número total de
                 ↪  amostras
                time = j / sample
                #Calcula o valor do sinal modulador para a amostra atual
                signal.append(I * cos(2 * pi * time) + Q * sin(2 * pi *
                 ↪  time))
```

```
        #Retorna o sinal modulado
        return signal
```

A modulação é dividida em duas funções, *qam_mapping* que recebe o sinal digital e o mapeia por trios de bits a valores correspondentes de amplitude, e também de fase, e a função *qam8_modulation*, que pega esses valores mapeados e calcula o valor da portadora a partir dos canais I e Q.

3.3 Eduardo Augusto Volpi

3.3.1 Camada de Enlace

O aluno implementou a camada de enlace em Python, abordando várias funções essenciais para garantir a transmissão confiável de dados entre dispositivos de rede. A implementação inclui técnicas de enquadramento, detecção e correção de erros, fundamentais para a integridade da comunicação digital. A seguir, são descritas as principais funcionalidades implementadas:

3.3.2 Enquadramento

O enquadramento é realizado utilizando o método de inserção de bytes (byte stuffing). Duas funções principais são responsáveis por esta tarefa: **Enquadrar:** A função *enquadrar insercao bytes* adiciona flags de início e fim ao quadro e insere bytes de escape quando necessário. Isso permite a delimitação clara dos quadros e evita a interpretação incorreta de dados que possam conter as sequências de flag ou escape.

```
def enquadrar_insercao_bytes(self, quadro):
resultado = self.FLAG
for char in quadro:
if char in (self.FLAG, self.ESC):
resultado += self.ESC
resultado += char
resultado += self.FLAG
return resultado
```

Desenquadrar: A função *desenquadrar insercao bytes* remove as flags de início e fim e interpreta corretamente os bytes de escape, recuperando o quadro original.

```
def desenquadrar_insercao_bytes(self, quadro):
if not (quadro.startswith(self.FLAG) and quadro.endswith(self.FLAG)):
raise ValueError("Quadro inválido: flags de início/fim ausentes")
resultado = ""
i = 1
while i < len(quadro) - 1:
    if quadro[i] == self.ESC:
        i += 1
    resultado += quadro[i]
    i += 1
return resultado
```

3.3.3 Detecção de Erros

Para a detecção de erros, o aluno implementou o uso do CRC-32 (Cyclic Redundancy Check): **Adicionar detecção de erro:** A função *adicionar deteccao erro* calcula o CRC-32 do quadro e o anexa ao final do mesmo.

```
def adicionar_deteccao_erro(self, quadro):
crc = zlib.crc32(quadro.encode())
return f"{quadro}|{crc:08x}"
```

Verificar erro: A função *verificar erro* recalcula o CRC-32 do quadro recebido e compara com o valor anexado, levantando uma exceção caso haja discrepância.

```
def verificar_erro(self, quadro):
dados, crc_recebido = quadro.rsplit("|", 1)
crc_calculado = zlib.crc32(dados.encode())
if f"{crc_calculado:08x}" == crc_recebido:
return dados
else:
raise ValueError("Erro de CRC detectado")
```

3.3.4 Correção de Erros

Para a correção de erros, foi implementado o código de Hamming: **correção de erro:** A função *adicionar hamming* aplica o código de Hamming a cada byte do quadro, adicionando bits de paridade.

```
def adicionar_hamming(self, dados):
resultado = ""
for char in dados:
codigo = self._calcular_hamming(ord(char))
resultado += ''.join(map(lambda x: chr(x + 48), codigo))
return resultado
```

Corrigir erro: A função *corrigir hamming* decodifica o quadro, corrigindo possíveis erros de um bit em cada byte.

```
def corrigir_hamming(self, quadro):
    resultado = ""
    for i in range(0, len(quadro), 12):
    codigo = [ord(c) - 48 for c in quadro[i:i+12]]
    if len(codigo) == 12:
        byte_corrigido = self._corrigir_hamming(codigo)
        resultado += chr(byte_corrigido)
    else:
        resultado += ''.join(map(chr, [c + 48 for c in codigo[4:]]))
    return resultado
```

3.3.5 Transmissão e Recepção

As funções *transmitir* e *receber* encapsulam todo o processo de preparação e processamento dos quadros: **Transmitir:** Esta função aplica sequencialmente o enquadramento, a detecção de erro, a correção de erro e, por fim, codifica o quadro em base64 para transmissão . **Receber:** Esta função realiza o processo inverso, decodificando o quadro de base64, aplicando a correção de erro, verificando a integridade com o CRC e finalmente desenquadrando o dado.

4 Conclusão

Este projeto de implementação de um simulador abrangendo as camadas física e de enlace do modelo OSI proporcionou uma compreensão aprofundada dos mecanismos fundamentais que garantem a transmissão eficiente e confiável de dados em redes de computadores. Através da

implementação em Python, foram exploradas diversas técnicas essenciais para a comunicação digital moderna.

Na camada física, foram implementadas múltiplas técnicas de modulação digital (NRZ Polar, Manchester e Bipolar) e modulação por portadora (ASK, FSK e 8-QAM). Cada uma das modulações oferece diferentes vantagens em termos de eficiência espectral, robustez ao ruído e facilidade de sincronização, demonstrando as considerações necessárias na transmissão de sinais digitais. Entretanto, houve dificuldade em definir as modulações por portadora para a modulação digital "Bipolar"em particular, pois esta possui três símbolos diferentes, 1, 0 e -1, em detrimento do nrz_polar e manchester que possuem apenas 2 símbolos. Dessa forma, não ficou claro o funcionamento das modulações ask, fsk e 8qam para três símbolos distintos.

A camada de enlace foi desenvolvida com foco em três aspectos críticos: enquadramento de dados, detecção de erros e correção de erros. O método de inserção de bytes para enquadramento, o uso do CRC-32 para detecção de erros e a implementação do código de Hamming para correção de erros exemplificam as estratégias empregadas para garantir a integridade e confiabilidade da transmissão de dados.

A integração desses componentes em uma interface gráfica utilizando Tkinter permitiu uma visualização clara e interativa dos processos de modulação e transmissão, facilitando a compreensão dos conceitos teóricos através de representações práticas.

Este projeto não apenas consolidou o entendimento teórico das camadas física e de enlace, mas também proporcionou experiência prática valiosa na implementação de protocolos de comunicação. As habilidades desenvolvidas em programação, análise de sinais e tratamento de erros são fundamentais para profissionais que trabalham com redes e sistemas de comunicação.

Em suma, este projeto demonstra a complexidade e a importância das camadas física e de enlace na comunicação digital, oferecendo uma ferramenta valiosa para o ensino e a pesquisa em redes de computadores e sistemas de comunicação.

5 Referências bibliográficas

- HUANG, Te-Yuan; JOHARI, Ramesh; MCKEOWN, Nick; TRUNNELL, Matthew; WATSON, Mark. A buffer-based approach to rate adaptation. **ACM Sigcomm Computer Communication Review**, [S.L.], v. 44, n. 4, p. 187-198, 17 ago. 2014. Association for Computing Machinery (ACM). http://dx.doi.org/10.1145/2740070.2626296.
- MAROTTA, Marcelo A.; SOUZA, Gustavo C.; HOLANDA, Maristela; CAETANO, Marcos F.. PyDash - A Framework Based Educational Tool for Adaptive Streaming Video Algorithms Study. **2021 IEEE Frontiers In Education Conference (FIE)**, Lincoln, NE, USA, 13 out. 2021. IEEE. http://dx.doi.org/10.1109/fie49875.2021.9637335.