Universidade de Brasília
Departamento de Ciência da Computação

TELEINFORMÁTICA E REDES 1:
TRABALHO FINAL

Hércules Natan Dantas de Almeida Medeiros
Leonardo Tomé Sampaio
Vinicius Gonçalves Duarte

1. Introdução

Neste trabalho, apresentamos a implementação e simulação das camadas de enlace e física, abordando conceitos fundamentais como modulação digital, portadora modulada, protocolos de enquadramento de dados, detecção e correção de erros. Para a transmissão do sinal, utilizamos os métodos NRZ-polar, Manchester e Bipolar, combinados com técnicas de detecção e correção de erros, como CRC-32, código de Hamming e bit de paridade, visando garantir a confiabilidade na transmissão das informações. A implementação foi desenvolvida em Python, utilizando o sistema operacional Linux e diversas bibliotecas, incluindo Numpy, Matplotlib, Python GTK, além dos módulos socket e threading para suporte à comunicação e paralelismo.

2. Implementação

2.1 Camada Física

Modulação digital

Non-return to Zero Polar (NRZ-Polar)

No protocolo NRZ-Polar, bits "1" são representados por tensões positivas e bits "0" por tensões negativas. O modulador não retorna à linha de base entre os pulsos, o que facilita a transmissão, mas limita a sincronização em sequências longas de "1" ou "0" devido à ausência de transições claras. Para contornar essa limitação, foi implementada a função `nrz_modulation(data: int) -> np.ndarray`.

Manchester

A modulação Manchester é um método de codificação de sinais digitais que garante sincronização por meio de transições no meio de cada bit. Nessa técnica, um bit '1' é representado por uma transição de nível alto para baixo (1 → 0) e um bit '0' por uma transição de baixo para alto (0 → 1), facilitando a detecção de bordas e o alinhamento temporal entre transmissor e receptor. No código, essa lógica foi implementada percorrendo cada bit da lista de entrada e substituindo-o pela sequência correspondente ([1, 0] para bit 1 e [0, 1] para bit 0), gerando assim um sinal final com o dobro de amostras e transições visíveis para cada bit.

Bipolar

Na modulação Bipolar (AMI), os bits ‘1’ são representados por tensões alternadas entre +1 e -1, enquanto os bits ‘0’ permanecem no nível zero. No código, isso é feito alternando o valor do bit 1 a cada ocorrência, usando uma variável **estado** para controlar a inversão do sinal, garantindo a alternância correta.

Modulação Portadora

Amplitude Shift Keying (ASK)

Na modulação ASK, a amplitude da portadora varia conforme o bit transmitido: bits “1” são representados por uma onda senoidal com amplitude alta, enquanto bits “0” geram um sinal com amplitude nula. No código, para cada bit é gerado um intervalo de tempo com 100 amostras, e o sinal resultante é construído aplicando $\sin(2\pi ft)$ com amplitude A para bits 1, ou zero para bits 0.

Frequency Shift Keying (FSK)

Na modulação FSK, a frequência da portadora muda conforme o bit: bits “1” utilizam uma frequência mais alta (f1), enquanto bits “0” usam uma frequência mais baixa (f2). O código implementa isso gerando, para cada bit, uma onda senoidal com a frequência correspondente, mantendo a mesma amplitude em ambos os casos.

8-Quadrature Amplitude Modulation

A modulação 8-QAM combina amplitude e fase para representar 3 bits por símbolo, totalizando 8 combinações distintas mapeadas para coordenadas (I, Q). O código converte a entrada binária em grupos de 3 bits, associa cada grupo a um ponto da constelação, e gera a onda modulada somando componentes $\cos(2\pi ift)$ e $\sin(2\pi ft)$. ponderadas por I e Q, produzindo um sinal com variação simultânea de amplitude e fase.

2.2 Camada de Enlace

Enquadramento

Contagem de Caracteres

Na técnica de **contagem de caracteres**, o primeiro campo de cada quadro indica quantos caracteres estão presentes nele. Isso permite que o receptor saiba exatamente onde o quadro termina, mesmo sem caracteres especiais de início ou fim. No projeto, essa lógica foi implementada pela função **contagem_caracteres(dado, tamanho)**, que converte o tamanho do quadro para uma representação ASCII binária (2 dígitos, totalizando 16 bits), insere esse cabeçalho no início do dado e, por fim, simula um possível ruído no canal ao aplicar a função de inversão aleatória de bits.

Inserção de Bytes

Na técnica de **inserção de bytes**, usa-se uma **flag especial** para marcar o

início e o fim de cada quadro, além de um **byte de escape** para evitar ambiguidades quando a própria flag aparece dentro do conteúdo transmitido. A função utilizada para esse método foi **insercao_bytes(dado, flag="01111110", escape="111111")**, que percorre os dados em blocos de 8 bits (1 byte) e adiciona automaticamente o byte de escape antes de qualquer ocorrência da flag ou do próprio escape. Depois disso, também é aplicado um ruído simulado com a função de alteração de bits, e o resultado final é encapsulado com a flag no início e no fim do quadro.

Detecção de Erros

Bit de Paridade Par

O bit de paridade garante que o número total de “1” na mensagem seja par, facilitando a detecção de erros simples. A função **bit_paridade(dado)** adiciona esse bit ao final do quadro. No receptor, a função **bit_paridade_receptor(dado)** verifica se o bit de paridade recebido é válido.

CRC-32

O CRC-32 aplica um cálculo polinomial para gerar um código de verificação de 32 bits, detectando diversos tipos de erro. A função **crc(mensagem_bits)** realiza esse cálculo e anexa o resultado. Já a **crc_receptor(mensagem_com_crc)** refaz o cálculo e valida a integridade da mensagem recebida.

Correção de Erros

Código de Hamming

O Código de Hamming é uma técnica que adiciona bits de paridade para detectar e corrigir erros simples na transmissão de dados. Na função **hamming_encod**e, calcula-se quantos bits de paridade são necessários, posiciona-se os bits de dados nas posições que não são potências de dois e os bits de paridade nas posições que são potências de dois, calculando seu valor com operações XOR para garantir a paridade correta. Já na função **hamming_encode_receptor**, as paridades são recalculadas para identificar a posição do bit com erro por meio da síndrome de erro; se um erro for detectado, o bit é corrigido invertendo seu valor, e em seguida os bits de paridade são removidos para retornar a mensagem original corrigida.

Fluxograma

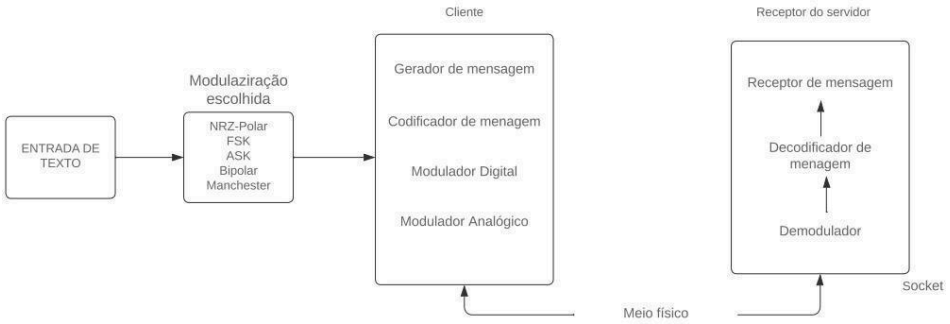

Figura 1: Fluxograma geral

O fluxograma ilustra o processo de codificação, transmissão, verificação e correção de erros.

1. **Cálculo e Inserção dos Bits de Paridade**:
    - Determina-se as posições dos bits de paridade.
    - Os bits de paridade são calculados para garantir que cada subconjunto de bits obedeça às condições de paridade.
2. **Formação do Quadro Codificado**:
    - O quadro é montado com os bits de dados e de paridade combinados.
3. **Transmissão do Quadro**:
    - O quadro é enviado ao receptor.
4. **Recepção e Verificação de Erros**:
    - Os bits de paridade são recalculados no receptor para verificar discrepâncias.
    - Caso os cálculos indiquem erros, a posição do erro é identificada com base nos bits de verificação.
5. **Correção ou Aceitação dos Dados**:
    - Se um erro for identificado, o bit correspondente é corrigido.
    - Caso não haja erros, o quadro é aceito como válido.
6. **Saída dos Dados**:
    - Após a correção (se necessária), os dados originais são recuperados e apresentados como saída.

O fluxograma apresenta o processo de codificação, transmissão, verificação e correção de erros

Membros

- [Hércules] - Implementação da camada de enlace, 8-QAM, interface e integração entre as camadas
- [Leonardo Sampaio] - Implementação da camada física e relatório
- [Vinicius Gonçalves] - Implementação da camada física e camada de enlace

Conclusão

Este projeto proporcionou experiência prática na conexão entre o cliente e o servidor usando sockets e threads. Aprendemos como criar um servidor que suporta conexões simultâneas e como combinar essa funcionalidade com a representação visual das mensagens recebidas.

O maior desafio é manter o servidor funcionando enquanto gerencia múltiplas redes e configura corretamente a troca de mensagens. Apesar das dificuldades, conseguimos um bom resultado: conseguimos construir um servidor que recebia constantemente mensagens dos clientes.